FEEDBACK BASED PERFORMANCE MANAGEMENT AND FAULT TOLERANCE FOR NETWORKED AND EMBEDDED COMPUTING SYSTEMS

BY

XUE LIU

B.S., Tsinghua University, Beijing, 1996
M.E., Tsinghua University, Beijing, 1999

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2006

Urbana, Illinois

Abstract

Performance management and fault tolerance are two important issues faced by computing systems research. In this dissertation, we exploit the use of feedback control for performance management and fault tolerance. Specifically, we propose Queueing Model Based Feedback Control scheme to achieve performance regulation. Traditionally, queueing theory was used for modeling computing system's performance. It usually serves as an offline capacity planning tool. On the other hand, feedback control theory was used for dynamically controlling the performance of electro-mechanical systems. How to utilize the "descriptive" power of queueing theory and the "prescriptive" power of feedback control to control computing system's performance was an open problem. Queueing Model Based Feedback Control answers this problem by integrating the strengths of both queueing model and feedback control into one unified framework. It provides better performance regulation compared with other control-theoretic approaches. We show the advantages of Queueing Model Based Feedback Control for two networked server applications: one is response time regulation for Apache Web server via dynamic resource allocations; the other is response time regulation for a multi-tiered Web service application via dynamic admission control.

In the second part of this dissertation, we further exploit the use of feedback control to achieve fault tolerance for real-time embedded control systems. We propose ORTGA (On-demand Real-Time GuArd), a new fault tolerance architecture which utilizes feedback control based software execution. ORTGA delivers the same functionalities as previously proposed Simplex architecture, with the same high fault coverage and reliability but with much more efficient resource utilization and flexibility. Hence it can be deployed in a wide range of

real-time embedded applications to provide fault tolerance. We implemented ORTGA in an inverted pendulum testbed to demonstrate its efficacy and efficiency. Based on the ORTGA design, we discussed the fault tolerance and scheduling co-design problem and its solutions.

To my beloved family

Acknowledgements

First and foremost, my deepest gratitude goes to my dissertation advisor, Professor Lui Sha. Professor Sha is truly a marvelous advisor, who is always dedicated to the success of his students. He not only provided invaluable guidance and support throughout my PhD study, but also taught me how to carry out creative research, how to approach complex problems, and how to build a successful career. I feel it extremely fortunate to work with Professor Sha, who is always an irreplaceable role model to me. The valuable training he provided will benefit my future career tremendously.

I would like to thank the honorable members of my dissertation committee, Professor Tarek Abdelzaher, Professor Jennifer Hou, and Professor P. R. Kumar. They have been very generous with their time providing insightful ideas, critical discussions, and valuable feedbacks on my work, many of which have been reflected in this dissertation. I am also deeply indebted for their advice and supports on my career choices.

During the past several years, I also had the honor to work with a line of distinguished researchers from both academia and industry, including Professor Marco Caccamo, Professor Chengyan Lu, Professor Ying Lu, Dr. Joseph Hellerstein, Dr. Yixin Diao from IBM Research, Dr. Xiaoyun Zhu, Dr. Sharad Singhal from HP Labs. Working with them has greatly benefited my research and career.

I am grateful for the help and support from all current and former members of the Real-Time Systems Group. Particularly, I would like to thank Jin Heo, Kihwal Lee, Hui Ding, Qixin Wang, and Rong Zheng for many insightful discussions and fruitful collaborations. I am also thankful to Chi-Sheng Shih, Sumant Kowshik, Tanya Creshaw, Ajay Tirumala,

Rodolfo Pellizzoni, Phanindra Ganti. Special thanks go to Ms. Molly Flesner for her wonderful administrative support. I very greatly appreciated her kindness and helpfulness.

I want to thank Songwu Lu, Martin Arlitt, Sujay Parek, Steve Froehlich, Chengdu Huang, Girish Baliga, Min Cao for their helpful discussions. I am indebted to UIUC and the department of Computer Science for providing me the environment to study and work in. I also want to thank various government and industry agencies, individual persons and organizations who provided support for my PhD study.

Last but not least, I want to thank my family members for their unconditional love and continuous support during my graduate study. Their love and support make my PhD journey a joyful experience.

I dedicate this dissertation to my family.

Table of Contents

List of Figures . . . . . **xi**

List of Tables . . . . . **xiii**

Chapter 1 Introduction **1**
1.1 Problems of Current Computing Systems 1
1.2 Feedback Control Reflection 3
1.3 Organization of Chapters 5

Chapter 2 Related Work and Motivations **7**
2.1 Feedback Control of Performance Management 7
2.2 Fault Tolerance 9
2.2.1 Simplex Architecture 10

Chapter 3 Queueing Model Based Feedback Control of A Single Computing Node **13**
3.1 Introduction 13
3.2 Conceptual Framework 15
3.2.1 Feed Forward and Feedback Controls 16
3.2.2 Control Model Development 19
3.3 Performance Evaluation 24
3.4 Experimental Investigation Using an Instrumented Apache Web Server . . . 28
3.4.1 Implementation 28
3.4.2 Monitor 30
3.4.3 Queueing Predictor 30
3.4.4 Controller 31
3.4.5 Connection Scheduler 31
3.4.6 Experimentation Results 32
3.5 Delay Variance Reductions 37
3.5.1 Motivations 37
3.5.2 Queueing Model Based Feedback Control with Queue Length Predictor 39
3.5.3 Effect of Queue Length Model Based Feedback Control 40
3.5.4 Evaluation 41
3.6 Conclusions 43

Chapter 4 Queueing Model Based Adaptive Control of Multi-Tiered Web Applications 44
4.1 Introduction and Motivation 45
4.2 Background 46
4.2.1 Muti-Tiered Web Application Architecture 46
4.2.2 Response Time Regulation via Admission Control for an Overloaded Web Site 47
4.3 Queueing Model Based Adaptive Control 48
4.3.1 Overview of the Queueing Model Based Adaptive Control Architecture 48
4.3.2 Queueing Predictor Design 50
4.3.3 Adaptive Feedback Loop Design 52
4.4 Experiment Setup and Implementation 55
4.4.1 Client Workload Generator 56
4.4.2 Controller, sensor and actuator implementation 57
4.4.3 Determination of other parameters 58
4.5 Experimental Results 58
4.5.1 Comparisons of Different Approaches under Workload A 59
4.5.2 Comparisons of Different Approaches under Workload B 62
4.6 Conclusions 63

Chapter 5 On-demand Real-Time Guard: A New Software Fault Tolerance Architecture 64
5.1 Introduction 64
5.2 Feedback Control of Software Execution for Software Fault Tolerance 68
5.3 ORTGA Design 71
5.3.1 Components Organization and Fault Recovery Procedure of ORTGA 72
5.3.2 CPU Usage Savings Analysis of ORTGA 73
5.3.3 Extra One Period Delay of ORTGA 74
5.4 Implementation and Evaluation of ORTGA 76
5.4.1 CPU Resource Savings of ORTGA 77
5.4.2 Fault Tolerance under ORTGA 79
5.5 Fault Tolerance and Scheduling Co-Design – Single FT-enabled Task Case . 84
5.5.1 Maximum Stability Region for Digital Controllers 84
5.5.2 Maximum Stability Region v.s. Control Period 89
5.5.3 Schedulability Analysis for ORTGA Fault Tolerance Architecture . . 90
5.5.4 Co-design of Fault Tolerance and Scheduling 96
5.6 Fault Tolerance and Scheduling Co-Design – Multiple FT-enabled Tasks Case 99
5.6.1 A Baseline Scheme for Multiple FT-enabled Tasks Case 99
5.6.2 A Hybrid Scheme for Multiple FT-enabled Tasks Case 101
5.7 Conclusions 103

Chapter 6 Final Remarks 104

Bibliography 106

Author's Biography **114**

List of Figures

1.1 Feedback control reflection. 4

2.1 The Simplex architecture. 11

3.1 Queueing Model Based Feedback Control architecture. 16
3.2 Linearization of $D(\mu) = 1/(\mu - \lambda)$ at delay reference = 2. 18
3.3 Effect of sample variance on model accuracy. 21
3.4 Effect of excitation on model accuracy. 22
3.5 Effect of control window size on sample variance. 24
3.6 Simulation results for performance evaluation. 26
3.7 Input load in case 1. 33
3.8 Exploring feedback control benefits. 34
3.9 Input load in case 2. 36
3.10 Exploring queueing predictor benefits. 36
3.11 Performance of Queueing Model Based Feedback Control under Pareto On/Off source. 38
3.12 Queue Length Model Based Feedback Control architecture. 40
3.13 Performance of Queue Length Model Based Feedback Control under Pareto On/Off source. 41
3.14 Comparison of two schemes using Web trace-driven simulation. 42

4.1 3-Tiered Web application architecture. 46
4.2 Online feedback based admission control architecture. 49
4.3 Architecture of Queueing Model Based Adaptive Control. 51
4.4 Adaptive feedback loop design using direct adaptive control. 55
4.5 Testbed infrastructure. 56
4.6 Performance comparison of Queueing model only, Adaptive control only and Queueing model + Adaptive control approaches under workload A. 60
4.7 Performance comparison of Queueing model + PI control and Queueing model + Adaptive control approaches under workload A. 61
4.8 Performance comparison of Queueing model + PI control and Queueing model + Adaptive control approaches under workload B. 62

5.1 A typical feedback control loop. 69
5.2 Feedback Control of Software Execution On Demand. 70

5.3 Illustration of the extra delay in recovery using ORTGA. 75
5.4 Illustration of the use of state projection to handle the extra delay. 75
5.5 Inverted pendulum control system protected by ORTGA. 77
5.6 Illustration of a stability region under state constraints. 87
5.7 Stability index v.s. control loop period. 90
5.8 Illustration of mode-change incurred by the recovery. 94
5.9 Illustration of the hybrid recovery scheme for multiple FT-enabled tasks case. 103

List of Tables

3.1 The aggregate error of different control schemes 35
3.2 Performance comparison using World Cup 98 traces 43

4.1 The aggregate errors under workload A 60
4.2 The aggregate errors under workload B 62

5.1 Execution statistics for HPC and HAC 78

Chapter 1

Introduction

Computing systems have now become an integral part of our information services infrastructure. There are many issues of computing systems faced by the research community and industry, such as performance (QoS), fault tolerance, scalability, safety and evolvobility. In this dissertation, we focus on two main issues, i.e. performance and fault tolerance. Though there are already extensive work addressing these two issues of computing systems, we discuss them from a different angle — using feedback control to provide performance management and fault tolerance. As we will show in this dissertation, feedback based approaches are very powerful and effective in performance management and fault tolerance for computing systems.

The rest of this chapter is organized as follows. In Section 1.1, we discuss the root cause of performance issues and fault tolerance issues in current computing systems and the importance for solving them. In Section 1.2, we motivate the use of feedback control for solving performance and fault tolerance issues in computing systems.

1.1 Problems of Current Computing Systems

As demands on functional and non-functional objectives of computing systems have continually increased for the last 30 years, so has the size of the resultant systems. Computing systems have become extremely large and complex, consisting of scores of software, hardware, and communications components [1]. Software-wise, this is a witnessed fact. For example, a

[1] We will focus on the discussion of software-related issues in this dissertation.

typical operating system in 1985, Tandem OS consisted of around 4 MLOC (Million Lines Of Codes). In 2001, the two typical operating systems, Linux and Windows XP, have around 30 MLOC and 40-50 MLOC respectively [19]. On the application software side, in 1998, the Apache project [98] source code is around 0.8 MLOC. This number increases to 10 MLOC in 2002, and continuously keeps increasing to 27 MLOC in 2004 [73].

The increasing software complexity poses two important issues to computing systems. First, due to the complexity, computing systems are becoming more poorly managed or maintained. Several factors contribute to this:

1. The increased complexity increases the skills level needed for the operators or administrators. As a result, there is a shortage of qualified operators for large complex computing systems IT-wide. Furthermore, the increased complexity also increases the cost of maintenance.

2. Because of the high complexity, many software systems expose a large number of "tuning knobs" for fine tuning the performance of the underlying systems. However, this makes it difficult for human operators to adjust the system in a real-time fashion when system working environment changes (such as workload), hence performance degradation may occur.

Poorly managed computing systems with degraded performance will affect the business profitability. For example, on World Wide Web where customers interact directly with Web servers, response time as a performance metric can have a direct and dramatic impact on business revenue. Forrester Research has published on InternetWeek [104] reporting: "One of the biggest reasons that people leave a site is because they're having a lousy experience — either pages aren't available or they take so long to download that it's just not worth it."

The second problem raised together with the increasing complexity is that complexity breeds bugs and errors in computing systems. Jim Gray estimated [19] that roughly there is 1 bug per KLOC (Kilo Lines Of Codes). Considering the code size of current softwares (in

the order of MLOC), the number of bugs could be very large. Software bugs may destroy any property that customers care about. These include the correctness, performance, reliability, security, and safety of the software.

Software bugs sometimes even bring the systems completely down [36]. From a company's point of view, the costs associated with software bugs can be staggering. For example, one prominent Internet company experienced an outage due to software bugs that lasted 22 hours, "resulting in a loss of revenue estimated at 3 million to 5 million dollars, with an associated 26% drop in its stock price [37]."

As we see, the ever-increasing software complexity makes it difficult to manage the performance of computing systems. Complexity is also a root cause for software bugs. Reports have shown that most giant software projects continuously face the battle with complexity. Unfortunately, they often fail. Complexity is becoming one of the major reasons that make software among the most poorly constructed, unreliable, and least maintainable technological artifacts invented by the mankind.

It is time that we need a paradigm shift to find an effective way to control the complexity of software systems. Can we make software systems smarter to manage themselves, hence can help solve issues related to complex software systems? In this dissertation, we exploit using feedback control to answer this question. Specifically, we focus on designing feedback based approaches for software performance management and fault tolerance.

1.2 Feedback Control Reflection

In this section, we motivate using feedback control to solve performance management and fault tolerance issues in computing systems.

Feedback control theory has been developed for more than 50 years. Many powerful tools and results have been obtained and deployed. Feedback control has been very successful in industry for electro-mechanical systems. With the help of modeling techniques [64],

control tools [9, 79] and digital computer implementations [10, 31], we now can manipulate the performance of electro/mechanical systems very well, even for very complex systems such as rockets and jet planes.

Computing systems have played an important role in the success of feedback control [10, 31]. Nowadays, computing systems are used in nearly every phases of control, including modeling, design, and implementations. However, the reverse is not true, as illustrated in Figure 1.1. Using feedback control for computing systems is just an emerging research area. A recent book by Hellerstein et. al. [35] gives an excellent summary of work in this new area.

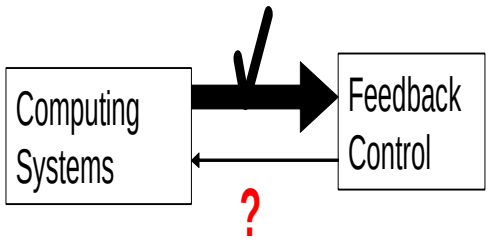


Figure 1.1: Feedback control reflection.

In fact, there are similarities between electro-mechanical systems and computing systems. Many of the issues need to be addressed in computing systems have their counterparts in electro-mechanical systems which are solvable using feedback control theories and tools.

In terms of organization and modeling, both computing systems and electro-mechanical systems usually have inputs and outputs; they also have internal dynamics within the system which relates the system input and output. In terms of management and control goals, one commonly used performance objective in the studying of computing systems is to ensure that system performance measure(s) are equal to or near the desired value(s). For example, to maintain the utilization of a server at 80% or to maintain the average client requests' response time at 0.2 second. These performance management problems map naturally to the **Regulation** problem [10, 31] in feedback control, in which the goal of the control is to ensure that measured output(s) are equal to or near the reference input(s).

Given these similarities between computing systems and electro-mechanical systems, it is natural that we apply feedback control to manage computing systems' performance.

Feedback control also has another property that is desirable for computing systems, i.e its robustness against errors [55, 79] such as model inaccuracies and measurement errors. As we have discussed in Section 1.1, how to tolerate software bugs and errors is another important issue in computing systems. In this dissertation, we will discuss using feedback control for both performance management and fault tolerance of computing systems.

1.3 Organization of Chapters

The remainder of this dissertation is organized as follows. We give a general overview and provide related work on feedback control of computing systems in Chapter 2, focusing on those related to performance management and fault tolerance. Chapter 3 presents Queueing Model Based Feedback Control framework for performance regulation[2]. We demonstrate the effectiveness of the proposed framework by controlling the average response time of Apache Web server using dynamic resource allocations. Next, in Chapter 4 we present Queueing Model Based Adaptive Control for admission control of a multi-tiered Web service application [3]. It combines the strength of both queueing models and adaptive feedback control. The purpose of this chapter is two-fold. First, we show that by using different combinations of queueing models and feedback control techniques, we can get a family of Queueing Model Based Feedback Control schemes suited for different applications and different QoS goals. Second, we show Queueing Model Based Feedback Control has wide application to autonomic performance management. It can be applied not only to single-node server systems, but also to general computing systems including multi-tiered Web applications. In Chapter 5, we present ORTGA (On-demand Real-Time GuArd), a new fault tolerance architecture for real-time embedded systems. It utilizes feedback control of software

[2]Our related publications contribute to this chapter are [62, 67, 94].
[3]Our related publications contribute to this chapter are [60, 61]

execution to provide fault tolerance for embedded control systems [4]. We demonstrate the effectiveness of ORTGA in an inverted pendulum control testbed. Finally we conclude the dissertation by laying out the future research directions in Chapter 6.

[4]Our related publications contribute to this chapter are [22, 59]

Chapter 2

Related Work and Motivations

Feedback control theory has been developed for more than 50 years. Many powerful tools and results have been obtained and deployed. Feedback control has been very successful in industry for controlling electro-mechanical systems. With the help of modeling techniques [64], control tools [9, 79] and digital computer implementations [10, 31], we now can manipulate the performance of electro-mechanical systems very well, even for very complex systems such as rockets and jet planes.

In this chapter, we will briefly review the work related to this dissertation. Since both performance management and fault tolerance are large research areas on their own, we could not give a detailed account here. Interested readers are referred to [5, 18, 71, 74] for reviews of a broader sense. In what follows, we will focus on those most pertaining to our work.

2.1 Feedback Control of Performance Management

Feedback control's application to computing system dates back to the 80's. It was embedded in the TCP congestion control protocol [39]. Since TCP was introduced to solve the congestive failure problems that had brought down the network, we have not experienced system-wide congestive failures again even the network has grown orders of magnitude. This is a testament of the effectiveness of feedback control in a highly dynamic, decentralized, and fast changing computing environment.

Recently, researchers discovered feedback control could be used in many modern computing applications, such as multimedia streaming, real-time computing, transaction pro-

cessing, embedded system, and e-commerce which require some form of performance control [4, 30, 66, 97, 102].

Chapter 1 in [35] gives an extensive summary of related work in this area. The systems being controlled include Lotus Notes email server [32], Apache Web server [4, 29, 66], Squid proxy server [70], Lustre file system [46], as well as sensor networks [2]. The output metrics include system-level metrics, such as CPU and memory utilizations [3, 29], cache hit ratio [69], and server queue length [80], or application level metrics such as response time and throughput [4, 46, 66]. Control mechanisms used include admission control or request throttling [46], Web content adaptation [3], application parameter tuning [29, 30], resource allocation [66, 69], and middleware [56, 106]. The types of control algorithms used include variations of proportional, integral, and derivative (PID) control [3, 4, 56, 80], pole placement [29], linear quadratic regulator (LQR) [29] , fuzzy control [56], and adaptive control [46, 69].

In order to use traditional feedback control framework, nearly all previous work use linear models to represent the underlying computing system. However, computing systems are usually nonlinear [51]. In addition, the workload may be stochastic and its parameters could change over a wide range of values. The differential/difference equation model used by traditional control theory does not model computing systems' performance well, except in the limited case of heavy workload that allows for fluid approximations.

A good-quality mathematical model of the plant is critical to any control system design. In spite of the tradition, we believe that the key to success of applying feedback control to manage computing systems' performance is not to force-fit a computing system into linear models. Rather, we should model computing systems as what they are.

During the last several decades, research has shown that queueing models represent computing systems well [40, 51]. Queueing model is usually used as an offline capacity planning tool in stead of an online performance tuning tool. In this dissertation, we try to answer the question that how we can combine the strength of both queueing models

and feedback control for performance management of computing systems. To this end, we propose Queueing Model Based Feedback Control framework. In Chapter 3 and 4, we will discuss Queueing Model Based Feedback Control framework in details. We will also show through real testbed implementations on different networked applications to validate the effectiveness of Queueing Model Based Feedback Control.

2.2 Fault Tolerance

Fault tolerance is always an important issue in software systems [71]. There are well-developed fault tolerance techniques for general software systems. They can be classified into three general categories: fault masking (e.g., N-version programming [11]), backward fault recovery (e.g., recovery blocks [85], checkpointing technique in transaction-based systems [84]), and forward fault recovery (e.g., roll-forward checkpointing scheme in [82]).

Our focus in this dissertation is to provide software fault tolerance for real-time control systems. None of the aforementioned schemes are readily applicable for real-time systems. In real-time systems, software faults can be categorized along three dimensions [92]: 1) resource sharing faults; 2) timing faults; and 3) semantic faults. The general fault tolerance schemes above do not sufficiently take timing faults into consideration. What is more, how to *timely* recover the system from faults in a real-time system has not been fully addressed in the general fault tolerance mechanisms.

Fault tolerance of real-time systems is an active research topic in the past decades. A comprehensive review of the recent progresses in this area can be found in [48, 74]. Here we only list the research which is most relevant to our work.

The FORTS project from University of Pittsburgh [12, 13] addresses the CPU scheduling of recovery jobs (not recovery processes or tasks) in real-time systems. One assumption of this project is that all faults can be detected, by whatever means, at the end of each periodic execution of the job, and the recovery is done by re-executing the job before its

deadline. In practical control systems, however, the fault of a controller process cannot be easily noticed until the effect of the bad control command shows up, i.e, there is a delay up to one or even more periods between when the fault occurs and when the control performance degradation or failure caused by the fault is noticed. What's more, usually a recovery cannot be done by simply re-executing the same job within the same period, which most probably will lead to the same fault since it is still executed within the same faulty controller process. A better approach can be replacing the faulty controller process with a high assurance controller process. A process replacing is different from a job re-execution and brings up many interesting research issues.

Lee et al. [53] develops a technique called *Process Resurrection* to recover a process from crash failure to meet the real-time requirements. Process Resurrection is tightly coupled with the crash detection mechanism of the underlying operating system, which offers signal as an event notification mechanism. Common error notification signals include SIGSEGV (segmentation faults), SIGBUS (bus error), SIGFPE (arithmetic operation error such as divided by zero), and SIGILL (execution of an illegal instruction). In Process Resurrection, a special signal handler is assigned for every crash related signals in order to override the default signal handler and trigger the recovery. However, the fault coverage of this technique is only limited to process crash.

2.2.1 Simplex Architecture

Simplex [89, 92] is a software architecture which facilitates the building of dependable real-time control systems. It provides dynamic toleration of system faults. In Simplex, the plant under protection is divided into a high-assurance-control (HAC) subsystem and a high-performance-control (HPC) subsystem. The HAC subsystem is a control software which was proven to be reliable. HAC's simple construction let the system designer leverage the power of formal methods and a rigorous development process. From the system level, high-assurance OS kernels such as certifiable runtimes are usually used in the HAC. From the

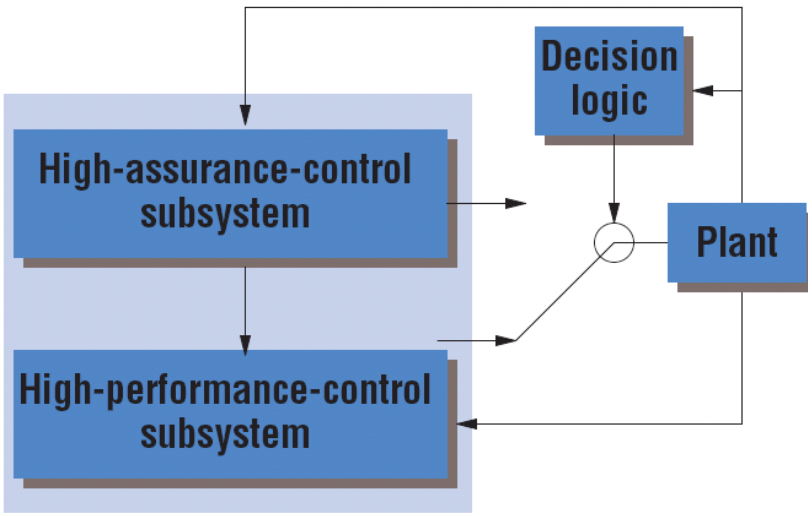


Figure 2.1: The Simplex architecture.

application level, well-understood classical controllers designed to maximize the controlled plant's stability region is also used.

The HPC subsystem complements the conservative HAC core. From the application level, an HPC can use more complex and advanced control technologies for higher control performance, including those difficult to verify, for example, neural network control. From the system level, COTS real-time OS and middleware designed to simplify the application development can be used in HPC. However, these software components may not be certifiable and could contain faults.

Figure 2.1 [92] shows the Simplex architecture. The HAC and HPC subsystems run in parallel, but the software stays in separate processes. Normally, the HPC controls the plant. The decision logic ensures that the plant state under the HPC stays within a HAC-established stability region. Otherwise, the HAC takes control.

As we can see from Figure 2.1, Simplex achieves fault tolerance by using feedback control of software executions. At every decision time, both the HAC and HPC get the state feedback from the plant and calculate their own control outputs. The decision module controls (determines) which output should be used for the plant. Then the plant will execute the control output accordingly. These feedback $\rightarrow$ control $\rightarrow$ execution steps constitute the

feedback control of software execution. By using the redundant HAC to guard against possible faults in the HPC, Simplex achieves fault tolerance.

However, several main drawbacks exist in the current Simplex architecture, as we will discuss in Section 5.1. In this dissertation, we design the new ORTGA (On-demand Real-Time GuArd) architecture (Chapter 5) which eliminates these drawbacks. ORTGA achieves the same high fault coverage and functionalities as Simplex, but with significantly lowered CPU resource usage and more flexibilities. We will discuss ORTGA and related research issues and associated solutions in Chapter 5.

Chapter 3

Queueing Model Based Feedback Control of A Single Computing Node

In this chapter, we present Queueing Model Based Feedback Control framework, which can help achieve better performance regulation for computing systems compared with previously proposed control-theoretic approaches. [1]

3.1 Introduction

As global E-Commerce continues to grow rapidly [20], the underlying architecture providing the service of E-Commerce is becoming increasingly important. The architecture is generally referred to as "Web services". The term "Web services" describes specific business functionality exposed by a company Web site through Internet connections, for the purpose of providing a way for another entity to use the services it provides [101]. Web services are the building blocks for the future generation of applications and solutions on the Internet.

One of the most pressing problems faced by Web services application designers and service providers is how they can provide the quality-of-service (QoS) required by their clients. Current practice typically relies on offline capacity planning to statically determine the resources to be allocated to ensure the QoS. However, Web traffic is highly dynamic and volatile [28, 76]. A carefully planned configuration for a Web site may work well under a specific traffic condition, but the same configuration may make the site go haywire when workload changes. A well-known example is the frequently-referred "Slashdot Effect" [6], which is named after the Web site slashdot.org. The "Slashdot Effect" occurs when a huge

[1] Our related publications contribute to this chapter are [62, 67, 94]

user base is referred to a previously undiscovered Web site which used to operate well. However, overwhelmed by the sudden increase in the traffic volume, the site's performance degrades or even crashes [44]. Since Web traffic is highly dynamic, offline capacity planning techniques are often not adequate for QoS control in Web applications. Instead, the system must be smarter to react to the changing workloads in an automatic way in order to maintain the desired performance.

Recently, control theory has also been successfully applied to controlling the QoS of computing systems. Chapter 1 of [35] gives an extensive summary of related work in this area. In order to facilitate the application of traditional control theory, most of the previous work on control-theoretic approaches uses linear difference (or differential) models to represent the underlying computing systems. However, computing systems are highly nonlinear [51, 75], except in the limited case of heavy workload that allows for fluid approximations.

A good model of the plant is critical to any control design. During the last 40 years, research has shown that queueing models serve as a fundamental tool to model computing systems [51, 52]. Queueing models have been successfully applied to areas such as capacity planning [75] and performance analysis [45, 51, 52, 75] etc. However, unlike feedback control, queueing theory usually is descriptive, rather than prescriptive. So its traditional application is in capacity planning rather than QoS control.

In this chapter, we try to answer the question of whether we can combine the strength of both queueing models and feedback control for performance management of computing systems. To this end, we present Queueing Model Based Feedback Control framework, a new framework for controlling the performance of computing systems. It utilizes the modeling (descriptive) power of queueing theory and the dynamic management (prescriptive) power of feedback control, hence it can give better QoS tracking than other previously proposed approaches.

In this chapter, we will first examine the case of timing performance regulation of a single-node Web server. In the next chapter, we will continue the discussion by examining

the case of multi-tiered Web applications.

The rest of this chapter is organized as follows. In Section 3.2, we first present Queueing Model Based Feedback Control on its modeling and control design in the context of a simple M/M/1 queue. We then generalize the approach to general computing systems which can be modeled by queueing models. Then we show the performance evaluation of Queueing Model Based Feedback Control via extensive simulations in Section 3.3. We use timing performance regulation of a Web server as a target application for Queueing Model Based Feedback Control. We present experimental evaluations of this approach using an instrumented Apache Web server in Section 3.4. We show that combining feedback control with a queueing model leads to better tracking of QoS specifications than with feedback control alone or queueing model based feed forward control alone. Hence, a major contribution of Queueing Model Based Feedback Control approach is the combination of feedback control theory and queueing theory to achieve more predictable timing behavior in computing systems such as networked servers. We present an improved method for further delay variance reduction under bursty traffics by exploiting server internal queue length information in Section 3.5.

3.2 Conceptual Framework

We use a Web hosting application as a motivating example of Queueing Model Based Feedback Control. From the perspective of a Web hosting company, we would like to keep the timing delay[2] experienced by users close to the service level agreement (SLA), e.g., an average delay of 1 second in a window of $1,000$ requests for a workload that is up to $100,000$ requests per second. Delays consistently longer than the specification are unacceptable to the users; Delays consistently shorter indicate over-provisioning of resources that could have been used for other users and applications.

[2]In this chapter, we do not distinguish the terms of "delay" and "response time". We use them interchangeably without ambiguity.

3.2.1 Feed Forward and Feedback Controls

In this section, we describe the fundamental elements of the Queueing Model Based Feedback Control. Classical results from queueing theory are used for computing the server service rate necessary to achieve a specified average delay given the currently observed average request arrival rate. Server resources are then allocated to achieve the computed service rate. We call this feed forward loop as the Queueing Predictor. Finally, a feedback control loop compares the actual delay achieved to the desired average delay reference and adjusts the resource allocation accordingly in an incremental manner to ensure that the desired delay is maintained. The feedback and queueing components operate concurrently in a complementary manner as shown in Figure 3.1.

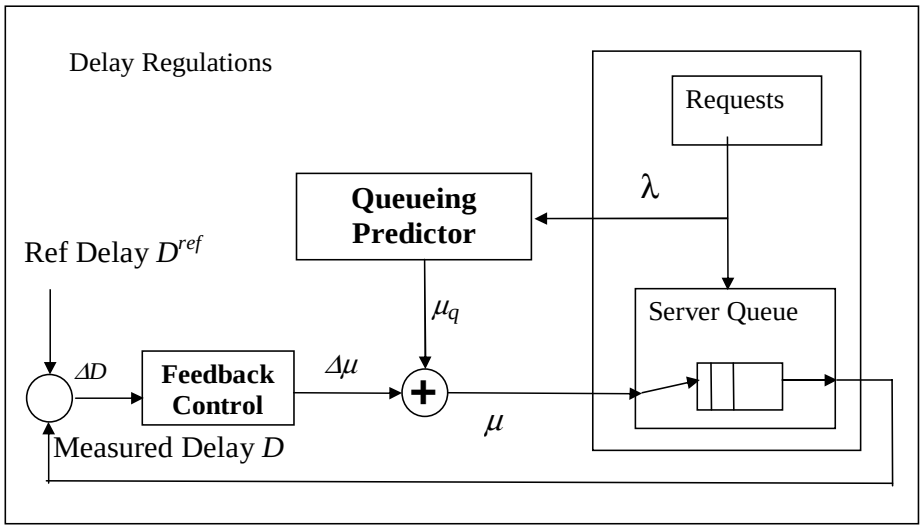


Figure 3.1: Queueing Model Based Feedback Control architecture.

In the following, we use a running example to explore the research issues that arise in the aforementioned scheme. We begin by illustrating the need for feedback control in a simple M/M/1 queueing model, which will be replaced by a G/G/1 model later. Modeling a Web server by a single queue is a commonly used simplification, because the performance a computer system is often dominated by a bottleneck stage.

Let the request process (shown at the top right corner of Figure 3.1) have arrival rate

λ, which may change abruptly. The queueing predictor estimates the arrival rate λ online. The equilibrium delay of an M/M/1 queue is simply $1/(\mu - \lambda)$ [40, 51]. Suppose the service level agreement with the users specifies that, over a desired window of events, the reference delay is $D^{ref} = 2$. Thus, the corresponding service rate $\mu_q = \lambda + \frac{1}{D^{ref}} = \lambda + 0.5$, which will give the long-term equilibrium delay of $D^{eq} = D^{ref} = 2$. However, because of the stochastic nature of requests, there may still be sizable fluctuations around $D^{ref} = 2$ and we wish to reduce the fluctuations within the user observation window. Thus, the mean sample delay d over a window will be computed and compared with the reference delay. The difference, Δd, is the error to be corrected by the feedback controller via service rate adjustment $\Delta\mu$ as shown in Figure 3.1.

Next, we design a simple feedback control loop to correct the error. We show how the presence of the Queueing Predictor makes a linear feedback controller sufficient by compensating for system non-linearity. For feedback control to improve system performance, the controller must be properly tuned. Figure 3.2 depicts the average queueing delay D of an M/M/1 model as function of $\mu - \lambda$. In addition, the reference delay was chosen to be 2 time units. The controller uses the error, ΔD, the difference between the measured delay from the desired delay reference to compute a adjustment in service rate, $\Delta\mu$, that would reduce the delay deviation. Hence, in order to design the feedback controller, we must know the effect of $\Delta\mu$ on Δd, which is known as the process gain $\Delta d/\Delta\mu$. For M/M/1 queue under a given arrival rate λ, we have $\Delta d/\Delta\mu = \Delta d/\Delta(\mu - \lambda)$. Thus, at the point of linearization, for small deviations, the process gain is approximately the slope of the non-linear rate-delay relation of the queueing system of $d = 1/(\mu - \lambda)$.

As suggested by the slope of Figure 3.2, changing resource allocation by the same amount will change the average delay by different amounts depending on the current service rate. In control-theoretic terms, a linearized controller for a non-linear system works well, only when the system state is in the neighborhood of linearization. The enforcement of this condition is achieved via the Queueing Predictor.

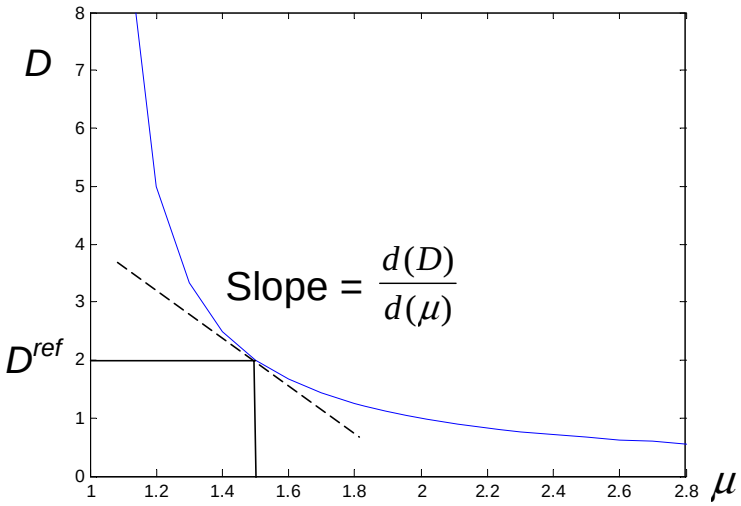


Figure 3.2: Linearization of $D(\mu) = 1/(\mu - \lambda)$ at delay reference = 2.

We illustrate by an example how the Queueing Predictor maintains the system in the neighborhood of linearization. The objective is to adjust $\Delta\mu$ to make the delay close to 2 time units, the reference delay.

Suppose that we linearize the system at $D^{ref} = 2$. Initially $\lambda = 1$. The feed forward control model provides $\mu_q = 1.5$. This generates an equilibrium delay $D^{eq} = 2$. The controller's task is to handle the sample mean fluctuations around 2. If the arrival rate suddenly jumps from 1 to 3, the queueing system moves far away from the point of linearization. Fortunately, as soon as the new arrival rate is detected, the feed forward service rate μ_q provided by the Queueing Predictor changes from 1.5 to 3.5 by solving the queueing model equation $D^{ref} = 1/(\mu - \lambda)$. As a result, the feed forward control (Queueing Predictor) restores the equilibrium condition to delay $D^{eq} = 2 = D^{ref}$. Thus, the feed forward control helps the feedback controller by keeping the system in the vicinity of linearization, no matter what the workload is.

In the following, we shall refer to the difference between the measured sample delay and the reference delay after the feed forward control as *residual error*. In Queueing Model Based Feedback Control, the purpose of the feedback controller design is to suppress residual errors around the target delay reference. The linear feedback controller not only can help us reduce

the fluctuations but also can correct inaccuracies in modeling and in the estimation of arrival rates. Having presented the basic ideas, we now raise the following questions.

- How can we accurately develop a linear model for the residual errors in a stochastic environment?
- How can we design a minimal order controller, and pick the correct rates of observation and control?

In the next subsection, we will answer these two questions.

3.2.2 Control Model Development

We now develop a general procedure that will allow us, based on experimental data alone, to get an accurate linear model for the residual errors in any queueing model based feed forward control.

We still use the M/M/1 queue as an conceptual model. When the output of the system is the average delay, the response time (average delay) formula, $1/(\mu - \lambda)$, depicts the average delay as a function of the difference between arrival rate and average service rate. From the perspective of control system modeling, this is exactly the transfer function between the control action and the response that we try to capture. Unfortunately, this function is valid only when the system can be modeled accurately by an M/M/1 model. We need a general procedure that works for any queueing model.

In our running example, as shown in Figure 3.2, the line tangent at the curve at $D^{ref} = 2$ has a slope of -4. This is because $dD/d\mu = -1/(\mu - \lambda)^2 = -(D^{ref})^2$. However, we want to rediscover it accurately from data of the residual errors alone. The standard approach in model identification [64] is to 1) use a white noise control signal that is sufficiently large but not causing saturation; and 2) rapidly sample the output. The collected data is then fed to least square based modeling tools such as the ARX model. The reason for the white

noise signal is that it will expose the full spectrum of system responses. The reason for large inputs is to make the responses larger and thus easier to measure. The reason for rapid sampling is not to miss any signal in the response due to the Nyquist rate consideration.

Unfortunately, the ideas of large excitation at the input and fast sampling at the output are the wrong things to do in the development of a control model for the residual errors in a queueing system, as we will discuss shortly. The correct approach is to use a small white noise signal, a large observation window and average all the observation samples in the window. This is counterintuitive. In common stochastic control problems such as Linear Quadratic Gaussian control problems and Kalman filters, the plant is fundamentally governed by differential equations, albeit random processes disturb both the observations and actuations. However, our plant here is a random process. And we want to control, in probability, the short-term behaviors of a random process by adjusting its probability distribution parameters.

To illustrate the need for a long observation window, suppose that we want to correct a biased coin with probability of head equals to 0.4. We begin by soldering a small weight to the side of head of the biased coin and then do some experiments. If the resulting probability of head is greater than 0.5, we will reduce weight. Otherwise, we increase it. Over a period of time, we may correct the biased coin. The critical question here is how frequently should we adjust the weight? Obviously, we cannot succeed if we change the weight, flip the coin once, and then change the weight again. Once we adjust the weight, we must flip the coin a large number of times until a statistical equilibrium is reached. This allows us to determine the effect of the added weight upon the probability distribution with a high degree of confidence. From a control perspective, the problem here is that we are controlling a probability, and not an instantly measurable quantity such as temperature or pressure. Since probability itself is not instantly measurable, the sensor can only estimate it by the mean of large samples. In other words, the transfer function between the change of weight and the change of probability of head manifests itself only when the sample variance becomes negligible.

Unlike a Bernoulli process where each coin flip is independent, the delays between nearby outputs from a queueing system are correlated. The sample variance drops and the mean of the window of events converge to the process mean as the window size increases and contains more and more epochs. This is the condition that allows the relationship between the control and its impact on delay to reveal itself.

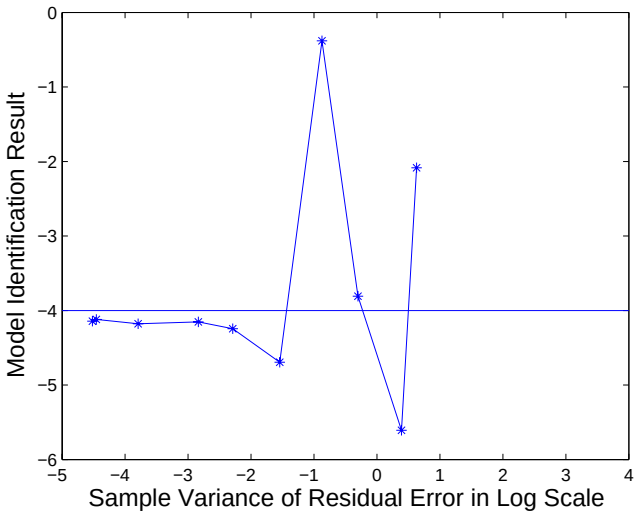


Figure 3.3: Effect of sample variance on model accuracy.

But there is one more important detail: we must keep the size of the excitation signal small. A close examination of Figure 3.2 tells us that the queueing delay curve is *asymmetric*. The effect of reducing $\Delta\mu$ has a greater impact than increasing it by the same amount, especially when the size of $\Delta\mu$ is large. This asymmetry causes problems in least square based estimation, because it will pick a line with equal sums of squared errors on both sides of the tangent. The need for small excitation also argues for a large sample size so as not to drown the responses to small excitations in sample variances.

We have implemented a simulator to study the performance of Queueing Model Based Feedback control whose architecture is shown in Figure 3.1. Our simulator is built on the SMPL simulation package [72]. In the simulation, we randomly change the sign of $\Delta\mu$ of a certain size and collected the sample means for model identification. Figure 3.3 depicts

the slope produced by Matlab's ARX model under a small excitation of $\Delta\mu = \pm 0.01$. The horizontal axis is the sample variance in log scale. As we can see, when the variances become very small (i.e., very large sample size), Matlab's estimation of the slope converges towards the theoretical value of -4. Figure 3.4 shows the effect of using a larger excitation of $\Delta\mu = \pm 0.1$. Due to the large excitation, the sample variance does not converge to *zero* as window size increases. In addition, the slope estimated by Matlab ARX model does not converges to -4.

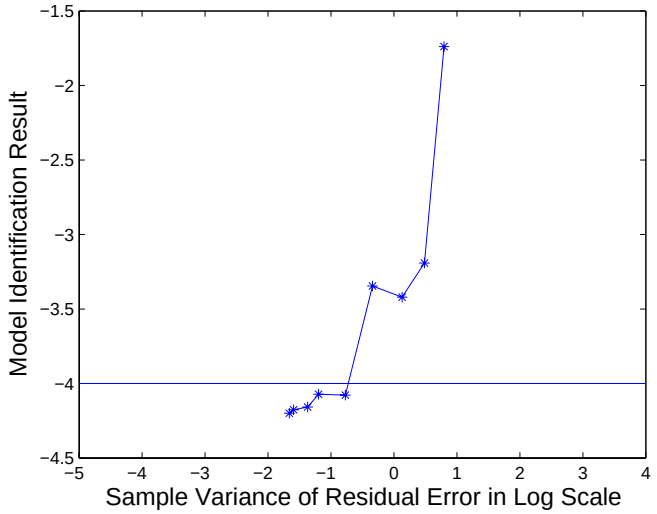


Figure 3.4: Effect of excitation on model accuracy.

One further fundamental distinction must be made in the choice of a sampling interval. In traditional control, the output of the process under control is a function of time. Thus, the unit of sampling is measured in units of physical time since they are the independent variables, upon which the process evolves. However, we now manipulate the probability distribution of a random process, where what counts is the mean and variance of delays. The independent variable is now the number of events that affects delay mean and variance, i.e., the number of served requests. Thus, it is preferable to perform our task in the domain of events rather than in that of time. This is also consistent with our event based model development process. Our sampling window is therefore measured as the number of elapsed

events.

We have so far developed a simple 1^{st} order ARX model that links the effect of increase (reduction) in service rate $\Delta\mu$ to reduction (increase) in output delays ΔD: $\Delta D(n) = -4\Delta\mu(n-1)$, where n is the n^{th} control sampling instance. The control gain k is simply -1/slope. That is, $k = 0.25$. This is because for next sampling instance of $n+1$, we want to correct the delay to $-\Delta D(n) = \Delta D(n+1) = -4\Delta\mu(n)$, so $\Delta\mu(n) = 0.25\Delta D(n)$. In control theory, this design corresponds to a deadbeat control design. We can also apply other control techniques, such as PI (Proportional Integral) controller, by treating $\Delta D/\Delta\mu$ as the transfer function between control input (service rate adjustments) and output (delay residual errors).

We now come to the final issue of designing the control rate. The observation window and the control window are often the same in many control applications. For example, when control period is 1 second, the signal used for the control is often based on what has been observed in this 1 second window. However, longer observation windows can be used. Consider the case where Kalman filter is used to condition the signal for a controller. Whatever the rate of control one chooses, the recursive nature of Kalman filter implies that at any time t, the signal used by the control is based on all the samples that have been observed from the beginning, albeit the effect of samples in the remote past is small.

For any chosen window of observation, we should update the control as soon as possible so as to minimize the build-up of errors. Suppose that the observation window contains 1000 events. We can produce the 1^{st} control update using the first 1000 events, the 2^{nd} update using the average of events 2 to 1001 etc. To summarize, our strategy is to use a long observation window but a short control update window. To reduce the computation overhead, we may choose to update once every N events instead of every event. Figure 3.5 shows the resulting control performance using an update window of 500 events, 200 events, ..., until 10 events. The horizontal axis depicts both the size of update windows and the corresponding size of the control effort, i.e., the average size of corresponding $\Delta\mu$. As we can

see, shorter update windows have two important effects. First, it reduces the sample mean fluctuations. Second, it also reduces the control efforts $\Delta\mu$. For examples, at N = 500, the average control effort is about 0.058. When $N = 10$, the average control effort is about 0.04. Smaller control effort makes the sharing of connections in server easier.

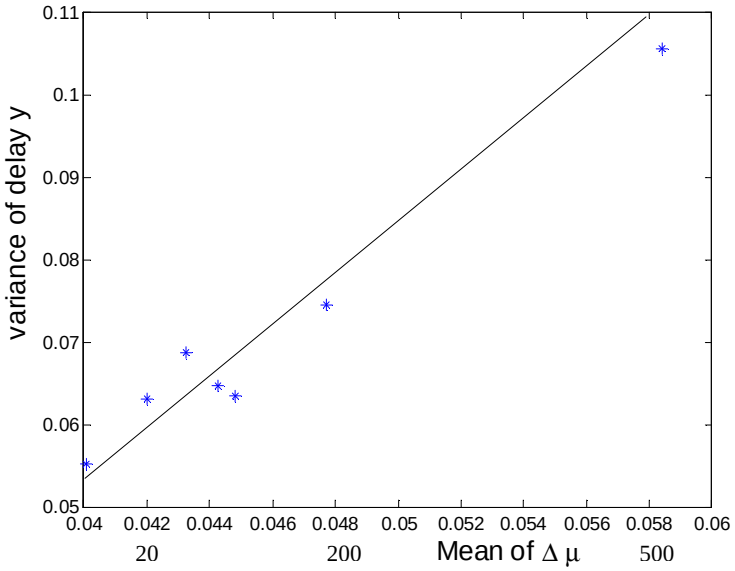


Figure 3.5: Effect of control window size on sample variance.

3.3 Performance Evaluation

Having introduced the basic concepts, we replace the M/M/1 model with a general G/G/k model, whose waiting time formula follows the Allen-Cunneen Approximation [16]:

$$D^q \cong \left(\frac{c_a^2 + c_s^2}{2}\right) \frac{\rho^{\sqrt{2(k+1)}-1}}{\mu \cdot k(1-\rho)}, \qquad c = \frac{s}{m}, \tag{3.1}$$

where traffic intensity is $\rho = \lambda/\mu$, and s and m are the standard deviation and mean of the sample data respectively. Subscript a and s stand for arrival process and service process

respectively. The steady state response time is

$$D^{eq} = D^q + 1/\mu. \tag{3.2}$$

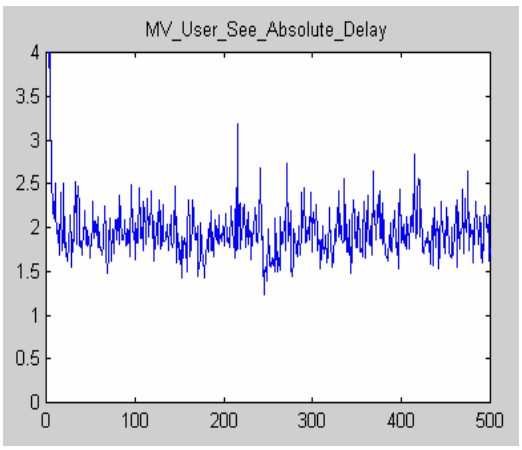
In the case of a server cluster with $k = 1$ node, we have

$$D^q \cong \left(\frac{c_a^2 + c_s^2}{2}\right) \frac{\rho}{\mu \cdot (1-\rho)} \quad . \tag{3.3}$$

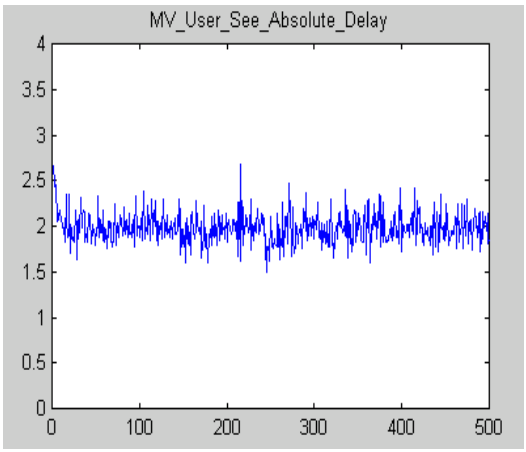
In our experiments, we replace the M/M/1 response time model with a G/G/1 response time model in the Queueing Model Based Feedback Control architecture shown in Figure 3.1. Both the mean and variance of the arrivals are now estimated online. For simplicity, we assume that the service time distribution is exponential but this can be easily replaced by online measurement of mean and variance as well. In the experiments, the reference delay is $D^{ref} = 2$. In the first set of experiments, the inter-arrival time has Pareto distribution, which was reported to fit measurements of actual Web traffic well [28]. In the beginning of the run, there are 0.65 arrivals per unit time. This rate jumps to 0.85 arrivals per unit time in the middle of the simulation. Figure 3.6(a) uses queueing model feed forward control only (i.e. only use Queueing Predictor for feed forward control).

The horizontal axis is the number of user observation windows with a size of 1000 events. The vertical axis is the sample mean of the delay in the user observation window. As we can see, feed forward control performs quite well by itself. This demonstrates the effectiveness of the queueing model. Figure 3.6(b) adds the additional feedback control to reduce the fluctuations around delay reference of $D^{ref} = 2$.

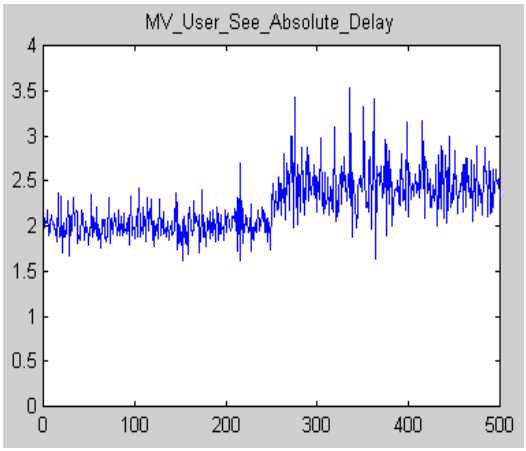
A precondition of a successful linearization of a non-linear system is that the point of linearization should be in an equilibrium state. The equilibrium service rate is the one that keeps the system in the equilibrium state. For a reference delay of $D^{ref} = 2$, we know that the equilibrium service rate is $\mu = (\lambda + 0.5)$. To reduce the random fluctuations around



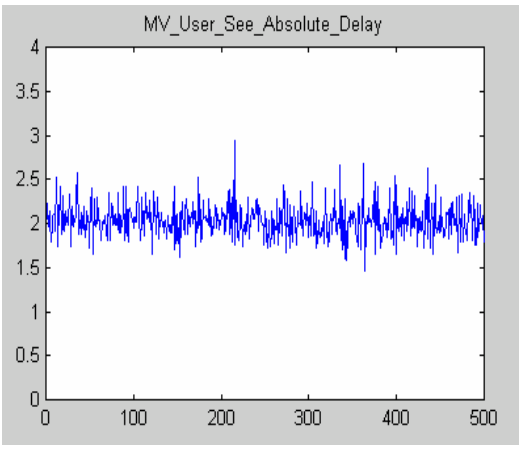
(a) Feed forward Control Only



(b) Feed forward + Feedback



(c) Control with Fixed Offset



(d) From Poisson Traffic to Pareto Traffic

Figure 3.6: Simulation results for performance evaluation.

$D^{ref} = 2$, the controller adjusts the service rate by a small amount according to the linearized model.

In Queueing Model Based Feedback Control, the equilibrium service rate is automatically provided by the feed forward control from the Queueing Predictor. A statically constructed equilibrium service rate is sensitive to the change in the arrival process. As we can see in Figure 3.6(c), a large steady state error and a larger variance materialize after the midpoint on the x-axis when the arrival changes.

Since a wrong equilibrium service rate results in a steady state error, we could use integral control to correct the equilibrium service rate. When a workload change causes steady state errors, the integral controller sums up the errors, and produces a negative feedback control that adjusts the service rate. However, until the set point is crossed, the sum of errors keeps increasing and so does the integral control. Thus, integral control has a tendency to produce overshoot. A small integral gain is preferred in practice. It allows a system to slowly correct steady state errors without excessive oscillations.

Unlike a load increase in a mechanical system that produces consistent steady state errors that can be directly canceled by integral control, a queueing system has sizable random swings around the process mean in steady state. From error history alone, it would take a long integration time to filter out random fluctuations and detect the steady state error caused by workload changes. This makes it difficult in practice to rely on the integral controller alone to provide the correct offset. On the other hand, a small integration term remains a useful tool to correct bias introduced by inaccuracies in the queueing model, such as the G/G/1 model used in our experiments.

Finally, to demonstrate the adaptivity of our approach, we begin with a Poisson arrival process with arrival rate 0.65 and then switch to a process whose inter-arrival time has a Pareto distribution with arrival rate 0.85. As we can see from Figure 3.6(d), Queueing Model Based Feedback Control is insensitive to the changes of distribution and rate of the arrival process.

3.4 Experimental Investigation Using an Instrumented Apache Web Server

In the previous two sections, we have discussed the conceptual framework of Queueing Model Based Feedback Control and evaluated its performance via simulation. The discussions and evaluations are based on single-queue queueing models. We also assume that we can control the service rate of the server accurately. In real-world applications, the real system may be different from the G/G/1 model we had used. Further, the control of service rate may not be able to be actuated directly. Given these model and actuation inaccuracies, we want to investigate how Queueing Model Based Feedback Control performs.

To this end, we implemented Queueing Model Based Feedback Control in an instrumented Apache Web Server. Apache is the most widely deployed Web server in the world. According to the survey conducted by Netcraft.com [78] on February 2006, there are around 51.8 million deployments of Apache Web servers on the Internet, which constitutes 68.1% of all the Web servers deployed when the survey was conducted. The control goal in our testbed is to keep the actual sample delays close to the specified delay reference by dynamic resource allocations using Queueing Model Based Feedback Control.

3.4.1 Implementation

We have conducted experiments of Queueing Model Based Feedback Control in an instrumented Apache Web Server [98] to evaluate the effectiveness of the aforementioned scheme in controlling the timing behavior of real-life applications[3]. When an Apache server boots up, a pool of processes is created to serve incoming TCP connection requests to the server. When an HTTP connection request arrives, it is put into a service queue to wait for an available process [4]. In HTTP 1.1, the latest HTTP protocol, persistent connections are

[3]The experiments were in cooperation with Professor Tarek Abdelzaher's group from University of Virginia.

[4]In some literature, these server processes are called worker processes.

implemented. Hence, a server process that accepts a connection must wait (i.e., block) for a specified amount of time for further requests from the same client. This blocking time dominates the time a process spends on serving the connection. Consequently, the average connection service rate is determined primarily by the number of server processes. The more processes are allocated to a client class queue, the higher the service rate. Thus, by adjusting the allocated processes, we can approximately control the service rate to the incoming requests and hence achieve the desired reference delay guarantees.

We modified the source code of Apache and added a new library that implements a Connection Manager that includes a Monitor, a Queueing Predictor, a Controller, and a Connection Scheduler. Let c denote the maximum number of server processes that the system could have (i.e. server capacity). Let $b(k)$ represent the number of server processes reserved for the given client class at the k^{th} sampling instant, where $b(k) \leq c$. In order to decide the process quota for the client class to be controlled, we implemented two modules, the Queueing Predictor and the Feedback Controller which correspond to the respective components in the Queueing Model Based Feedback Control architecture as shown in Figure 3.1. According to the resulting process quotas, the Connection Scheduler serves as an actuator to allocate certain number of server processes to connections from this client class.

The Connection Manager process runs a high-priority loop that listens to the Web server's TCP socket, accepts incoming connection requests, and queues them up in the application layer. This design allows us to control the request queue without kernel modifications. The Connection Scheduler module dispatches a connection by sending its descriptor to a free server process through a corresponding UNIX domain socket. The Connection Manager time-stamps the acceptance and dispatching of each connection. The difference between the acceptance and the dispatching time is recorded as the connection delay that models the service delay we are concerned within this study. Although the service delay also includes the queueing time in the TCP listen queue in the kernel, in our prototype this kernel-level queueing delay is negligible by design. This is because the Connection Manager always

greedily accepts all incoming connection requests in a tight loop.

3.4.2 Monitor

At each sampling instant k, the Monitor is invoked to compute the average request arrival rates $\lambda(k)$, mean request inter-arrival time $m(k)$ and its standard deviation $s(k)$, during the last sampling period. They are used by the Queueing Predictor to calculate new process quotas $b(k)$ as discussed for the G/G/1 model in Section 3.3. The Monitor also computes the average connection delays $W(k)$ of the client class during the last sampling period, which will be used by the Feedback Controller to calculate adjustment $\Delta b(k)$ for process allocations.

3.4.3 Queueing Predictor

System profiling was carried out beforehand to get an approximate value for μ, the service rate per process unit. The Connection Manager time-stamps the dispatching and closing of each connection. The difference between the dispatching and the closing time is recorded as the service time the server process spending on the connection. Using the average service time of server process per connection, we calculated an estimate of the service rate μ of one server process.

As discussed in Section 3.3, we assume a G/G/1 queueing model for the Apache Web server. For simplicity, we assume that the service time distribution is exponential. Our experimental results presented below can be further improved if a better estimate of the distribution of service times is available. In practice, however, accurate online measurement of this distribution may be both difficult and unnecessary. At each control period, the Queueing Predictor will use the estimated service rate μ, the online measured average request arrival rates $\lambda(k)$, the average request inter-arrival time $m(k)$ and its standard deviation $s(k)$ to produce the new desired values for $b(k+1)$ by solving equation (3.3).

3.4.4 Controller

Both a P (Proportional) and a PI (Proportional Integral) controller were implemented to control adjustment of server process quota. Both controllers are variations of the general PID (Proportional Integral Derivative) controller [8,55] known for its robustness and predominant use in industry. The derivative control was not used, because it is sensitive to noises in measurements. The advantage of a P controller is simplicity. The advantage of a PI controller is better elimination of residual errors. The Monitor computes and passes the average delay to the Feedback Controller, which computes the error $e(k)$ between the measured delay $D(k)$ and the desired delay D^{ref}. The output $U(k)$ of the P controller is simply proportional to the error. A digital form of the PI control function is:

$$U(k) \;=\; U(k-1) \;+\; g(e(k) - r \times e(k-1)),\; U(0) \;=\; 0, \tag{3.4}$$

where $U(k) = \Delta b(k)$, the adjustment of the server process quota; g and r are design parameters called the controller gain and the controller zero, respectively (see [66] for the parameters design using Root Locus method).

3.4.5 Connection Scheduler

The Connection Scheduler serves as an actuator to control the connection delays of the client class. It adds up the outputs from the Queueing Predictor and the Feedback Controller and computes the individual values of process allocation $b(k)$ for the client class. The Connection Scheduler listens to the well-known port and accepts every incoming TCP connection request. The Connection Scheduler maintains a (FIFO) connection queue Q and a process counter R for the number of processes allocated to a connection.

3.4.6 Experimentation Results

In this section, we present experimental results for our instrumented Queueing Model Based Feedback Control Apache Web server, in which the Queueing Predictor and the Feedback Controller are integrated.

Experimental Setup

All experiments were conducted on a testbed of PCs connected with 100Mbps Ethernet. One machine with a 450MHz AMD K6 processor and 256MB RAM was used to run the Web server with HTTP 1.1 enabled, and two machines with 450MHz AMD K6 processor and 256MB or 64MB RAM were used to run clients that stress the server with a synthetic workload. The experimental setup is as follows.

Client

We modified SURGE (Scalable URL Reference Generator) [28] to generate synthetic Web workloads in our experiments. The current version of SURGE assumes that users do not issue another request until they get the reply for the previous one. This assumption does not model well the behavior of a server with many independent clients whose request arrival times are not related to the times responses were sent to other requests. Hence, we modified SURGE to generate URL requests with a rate independent of the server load while still keeping the self-similarity [27, 28] characteristics in the resulting traffic.

Server

We implemented the Queueing Predictor and Feedback Controller in the instrumented Apache Web server [98]. The maximum number of server processes is configured to $c = 128$, which models a small size Web server. Each sampling period is 600 events long in all the experiments. To obtain a consistent system model from one sampling period to another, we fix the number of events (trials) per sample. The connection TIMEOUT of HTTP1.1 is set to 15 seconds.

Experimental Results

In order to evaluate the performance of the Queueing Model Based Feedback Control approach, we compared its performance with that of the feedback controller only and Queueing Predictor only systems respectively. The goal of the system is to provide the service delay guarantee for the client class with as few resources as possible while meeting the performance requirement. In practice, it is common for the Web service provider to promise the premier class certain performance guarantee while at the meantime, trying to provide as good performance as possible to the other classes. Therefore, it is desirable that only the necessary amount of resource is allocated for the premier class. Two experiments were conducted, where the service demands were changed dramatically. In both experiments, the desired connection delav for the class is set to $D^{ref} = 2$ seconds.

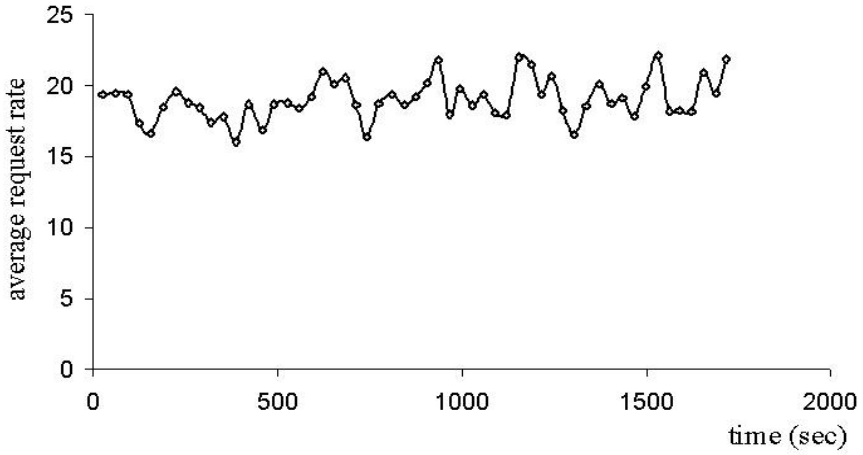


Figure 3.7: Input load in case 1.

The input load patterns in the two experiments are shown in Figure 3.7 and Figure 3.9 respectively. The collected average connection delay measured in every sampling period (i.e. every 600 events) are shown in Figure 3.8 and Figure 3.10 respectively. [5]

In the first experiment, we explore the effect of adding feedback control to a resource allocation scheme that uses a G/G/1 predictor alone for resource allocation. Figure 3.8 shows how tightly the target delay is tracked by the system, comparing the performance

[5]Data and figures are reported in our paper [94].

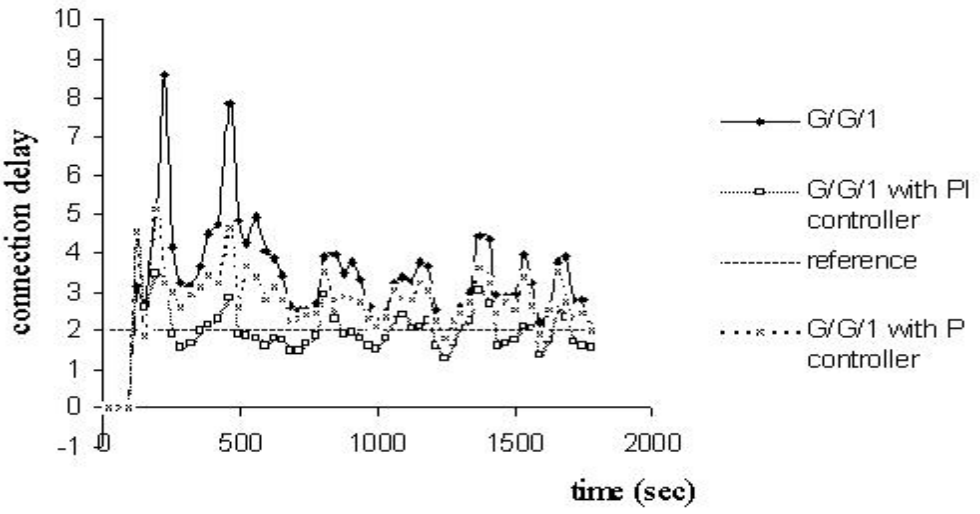


Figure 3.8: Exploring feedback control benefits.

of the G/G/1 predictor alone to that of the predictor augmented with a P and then a PI controller (i.e. Queueing Model Based Feedback Control). In the second experiment, we explore the benefits of adding the predictor to a resource allocation scheme that relies on feedback control alone. Figure 3.10 compares how well the delay target is achieved in the feedback control scheme with and without the G/G/1 predictor.

The experiments show that the Queueing Model Based Feedback Control scheme outperformed both the queueing predictor and the feedback control scheme used in isolation. Experimental results reveal the following observations:

- Using the aggregate of the squared errors between the target and actual connection delay over the duration of the experiment, we can compare the performance of different schemes. The smaller the aggregate error, the better the convergence. The following table summarizes the results.

- Using queueing model feed forward control alone, a large steady state error develops (Figure 3.8). This is attributed to the fact that the Web server is not exactly a single queue system. Thus, the model derived in this chapter is only an approximation of the true server. The fluctuation is also quite large.

Table 3.1: The aggregate error of different control schemes

Experiment 1 (Figure 3.8)	G/G/1 predictor alone	G/G/1 predictor with a P controller	G/G/1 predictor with a PI controller
Aggregate error	214.08	74.80	30.32
Experiment 2 (Figure 3.10)	PI controller alone	PI controller with a G/G/1 predictor	
Aggregate error	56.84	23.84	

- When P (proportional) control is added, both the steady state error and fluctuations decrease (Figure 3.8). Proportional controllers are not effective for correcting steady state errors caused by load disturbance, because they need the existence of instantaneous errors to generate corrections.

- When integral control is added (feed forward plus PI control), the steady state error is eliminated (Figure 3.8).

- The closed loop performance of the PI controller is much better in the presence of the queueing predictor (Figure 3.10). This is because the predictor is able to supply an approximate output that achieves a delay value close to the set point. The controller therefore has to handle the residual error only.

Our experimental evaluation demonstrates the advantages of integrating a queueing theoretic predictor with a feedback controller to achieve QoS guarantees in networked servers. The two components have complementary strengths, jointly offering more robust tracking of performance set points in the presence of widely unpredictable load.

While queueing theory and feedback control have been repeatedly used in isolation in many contexts to provide performance guarantees, we are not aware of any prior work that demonstrates and advocates their combined use within a single framework. We believe that such a combined framework merits deeper investigation to produce better theory for performance assurances in computing systems.

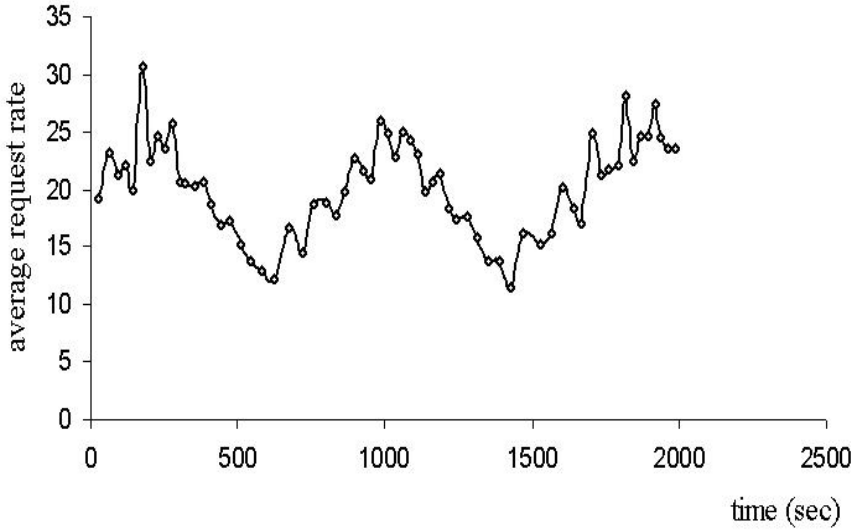


Figure 3.9: Input load in case 2.

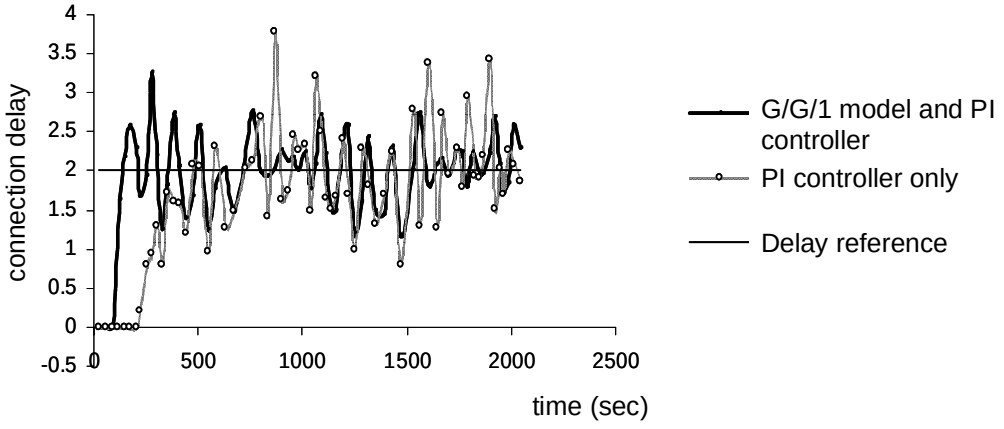


Figure 3.10: Exploring queueing predictor benefits.

3.5 Delay Variance Reductions

Queueing Model Based Feedback Control can achieve good delay regulations. However, we still see relative large response time variations. In this section, we improve Queueing Model Based Feedback Control by exploiting the use of server internal queue length measurements. The resulting Queueing Model Based Feedback Control with Queue Length Predictor (we simply call it "Queue Length Model Based Feedback Control" in the following discussions) allows better handling of the transient behaviors caused by rapidly changing Web traffic loads. As a result, it helps reducing response time variance. Queue Length Model Based Feedback Control can be regarded as an extension of Queueing Model Based Feedback Control.

3.5.1 Motivations

Studies reveal that the Web traffic is bursty and exhibits self-similar properties [27] . It has been shown the burstiness of Web request traffic can be modeled using the Pareto On/Off distribution. In this model, packets are sent at a fixed rate during On periods, and no packets are sent during Off periods.

To evaluate the performance of different control approaches when the traffic is bursty, we extended the request generator in our simulator (Section 3.2.2) to include Pareto On/Off traffic. In the simulation, there are three adjustable parameters to control the "burstiness" of the Pareto On/Off traffic. The parameter *Burst_Time* corresponds to the mean length of On period of the traffic. The parameter *Idle_Time* corresponds to the mean length of the Off period and *Interval* is the inter-arrival time during On periods. In all the simulations, the Pareto shape parameter is set to 1.5 as evidenced from real traffic measurements [38].

Figure 3.11 shows the delay experienced by a single client connection with *Burst_Time=1*, *Idle_Time =10* and *Interval*=0.1 under the original Queueing Model Based Feedback Control. The reference delay is set to $D^{ref} = 2$. Each point in the upper figure is an average of

response times experienced by 1000 requests. The corresponding mean and variance of the response time are 2.0041 and 1.2034 respectively. The lower half of Figure 3.11 depicts the time of request arrivals, which clearly shows the burstiness of the arrival traffic.

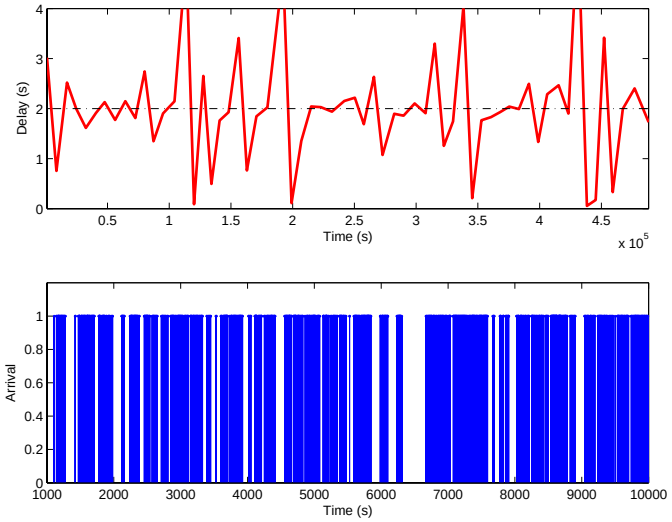


Figure 3.11: Performance of Queueing Model Based Feedback Control under Pareto On/Off source.

From Figure 3.11, we observe that under bursty traffic the delay fluctuates around the reference value although the long-term average mean delay is close to D^{ref}. This is because the online estimation of the request rate λ is a time average of the instantaneous request rate (on the order of 500 requests in our implementation). For bursty traffic, during the On periods, the instantaneous request rate is larger than the long-term average rate λ. However, the feed forward queueing predictor's output rate is based on λ and thus is lower than the instantaneous request rate. Therefore, the request queue will build up and clients will experience longer response time. On the other hand, during periods with sporadic requests, the instantaneous request rate is smaller than λ which leads to smaller response time. Since the variance in delay is correlated with the queue length, this observation motivates us to consider the use of server internal queue length information in the controller design to suppress large delay variations.

3.5.2 Queueing Model Based Feedback Control with Queue Length Predictor

To have a better control of the transient behavior, we utilize queue length information in our new controller design based on Queueing Model Based Feedback Control. In fact, queue length is closely related to the delay of a request, which equals to the sum of service times of all requests queued ahead of it and the service time of its own.

First, we introduce a new *Queue Length Model Based Predictor* with an additional feedback term, i.e., the queue length measurements of server in the prediction of service rate allocation μ_q. The procedure of Queue Length Model Based Predictor is as follows:

Step 1: At each control invocation, we measure the current queue length $l_{current}$ and update the request rate estimate λ.

Step 2: Based on results from queueing theory, a targeted queue length $l_{targeted}$ is computed. For example, if M/M/1 (or G/G/1) model is used to model the server, we have $l_{t\arg eted} = \lambda D^{ref}$. The term $l_{targeted}$ gives the desired queue length in steady state under the current mean request rate λ and targeted delay reference.

Step 3: Let $\mu_q = (\frac{1}{D^{ref}} + \lambda) + K \times (l_{current} - l_{t\arg eted})$ be the new model output service rate (i.e. feed forward service rate). The first term $(\frac{1}{D^{ref}} + \lambda)$ is the same as the Queueing Predictor in the original Queueing Model Based Feedback Control approach. The second term $K \times (l_{current} - l_{targeted})$ represents the queue length feedback. K is a constant control gain, in practice, we can set $K = 1/D^{ref}$.

In essence, this new queue length model based predictor adjusts the estimated service rate μ_q not only based on the request rate estimation but also on the degree of server queue built-up. From discussions in Section 3.5.1, if the request rate in the current control interval is larger than the mean request rate estimation λ, we have $l_{current} > l_{targeted}$. Therefore, the second term ("Queue Length Predictor") is positive in Step 3, which helps to clear up the queue and thus reduces the client experienced response time.

The proposed Queue Length Model Based Feedback Control architecture is shown in Figure 3.12. In this architecture, the Queue Length Model Based Predictor outputs a service rate allocation μ_q based on the online measurement of request rates, reference delay D^{ref} and current queue length $l_{current}$. The feedback controller calculates a service rate adjustment $\Delta\mu$ according to the difference between delay reference D^{ref} and measured delay D in each control interval. Lastly, the sum of μ_q and $\Delta\mu$, i.e, μ is used to determine the system resource quota to be applied to the server.

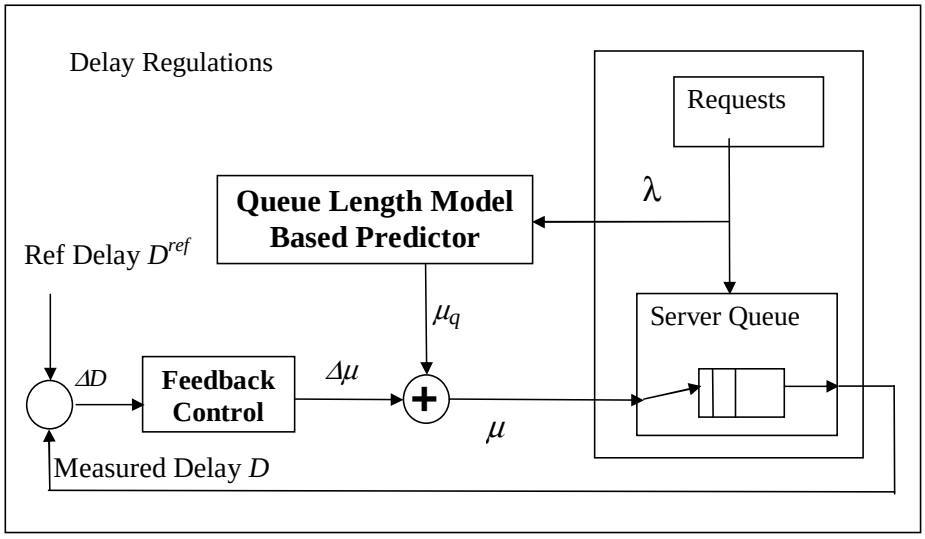


Figure 3.12: Queue Length Model Based Feedback Control architecture.

3.5.3 Effect of Queue Length Model Based Feedback Control

We implemented the Queue Length Model Based Feedback Control in our simulation package. To verify the design of the proposed controller, we experiment with the same Pareto On/Off traffic (with parameters $Burst\_Time = 1$, $Idle\_Time = 10$ and $Interval = 0.1$) as in Section 3.5.1. Figure 3.13 demonstrates the controlled server performance using the new Queue Length Model Based Feedback Control. The variance of the client response time reduces to 0.0051 as compared with 1.2034 using the previous Queueing Model Based Feedback Control approach. In addition, the average response time of both approaches are very

close to $D^{ref} = 2$. We see using the new controller, even with this extremely bursty traffic, good delay regulation can be achieved.

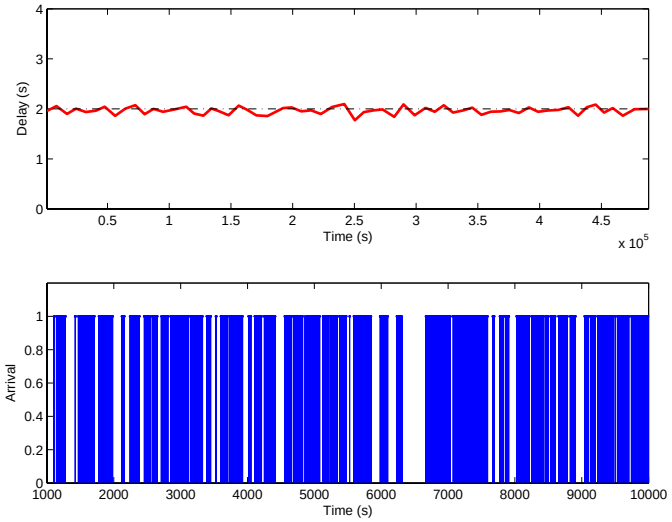


Figure 3.13: Performance of Queue Length Model Based Feedback Control under Pareto On/Off source.

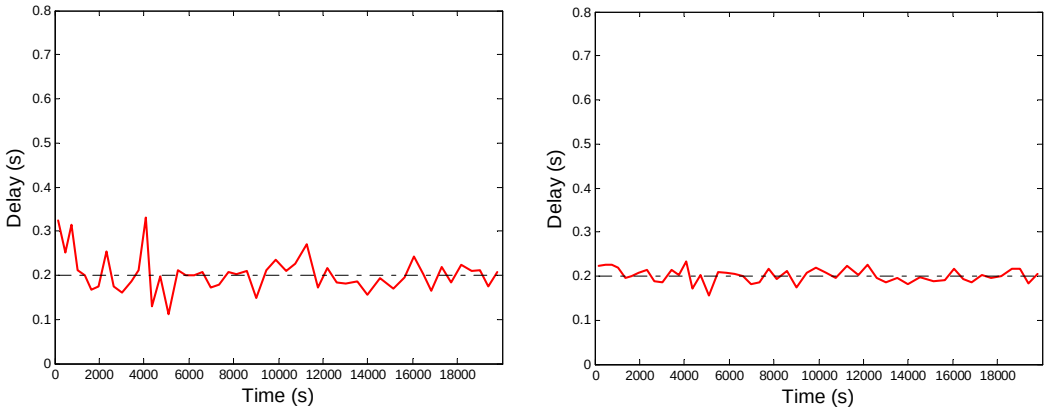
3.5.4 Evaluation

In this section, we evaluate the performance of Queue Length Model Based Feedback Control using real-life Web server traces. Experiments are performed using the World Cup 98 Web trace [7].

Trace-driven Simulation

First, we evaluate the performance via simulation. We extend our simulator to use real-world Web trace for generating client requests. Figure 3.14 shows the simulation result using the world cup trace as client requests.

We see both schemes perform well by regulating the delay close to the delay reference, but by adding the additional queue length predictor, Queue Length Model Based Feedback Control incurs lower variance than Queueing Model Based Feedback Control.



(a) Queueing Model Based Feedback Control

(b) Queue Length Model Based Feedback Control

Figure 3.14: Comparison of two schemes using Web trace-driven simulation.

Experimental Results

We have implemented Queue Length Model Based Feedback Control on Apache web server 2.0.7 with Linux kernel 2.4.20. In our implementation, Apache and Linux kernel are modified to provide state information (such as queue length) used in the control system.

All experiments are conducted on a testbed consisted of two PCs connected through 100Mbps Ethernet. The client machine is equipped with a 1.7GHZ Intel Pentium IV processor and 512MB RAM. httperf [41] is used as synthetic client request generator on the client. We modified httperf to generate realistic workload from real Apache access log (i.e. web traces explained below). The server machine has a 333MHZ Intel Pentium II processor and 256MB RAM, which runs Apache 2.0.7 and Linux kernel 2.4.20.

Experiments are performed using the World Cup 98 Web trace [7]. For the World Cup 98 trace, inter-arrival time is calculated between consecutive requests and scaled by different scale values among 1.0, 0.5, 0.3, 0.2, such that

$$Adjusted\_Interarrival\_Time = Interarrival\_Time\_from\_the\_Trace \times Scale\_Value$$

In all experiments, the reference delay is set to 0.1 (seconds).

We compare the performance of original Queueing Model Based Feedback Control and

Queue Length Model Based Feedback Control (i.e. with queue length predictor) under the world cup trace workload. The mean and variance of delay measurements collected from the testbed are summarized in Table 3.2.

From Table 3.2, we observe that Queue Length Model Based Feedback Control can regulate the delay better than the plain Queueing Model Based Feedback Control. The response time is very close to the reference delay of 0.1 second. Furthermore, the variance of response time is also smaller than the approach when there is no queue length predictor added. So our recommendation is when server internal queue length is available, we can further improve the performance of Queueing Model Based Feedback Control by adding the queue length predictor in to suppress delay variance, especially when the traffic is bursty.

Table 3.2: Performance comparison using World Cup 98 traces

Inter-arrival x	Queueing Model Based Feedback Control		Queue Length Model Based Feedback Control	
	Mean(RT)	Var(RT)	Mean(RT)	Var(RT)
0.3	0.089887 s	0.005249	0.105355 s	0.000274
0.2	1.149359 s	0.285477	0.109762 s	0.000337

3.6 Conclusions

This chapter introduced a new performance regulation framework — Queueing Model Based Feedback Control. It can track the QoS for computing systems in unpredictable environments. Queueing Model Based Feedback Control integrates components from both queueing theory and feedback control. We discuss fundamental issues and problems that arise in implementing this framework. Both simulation and experimental results demonstrated the effectiveness of the proposed framework. We also propose adding queue length feedback to further reduce the output variance. The resulting Queue Length Model Based Feedback Control can help to reduce variance but still achieve good regulation, especially when the workload is bursty.

Chapter 4

Queueing Model Based Adaptive Control of Multi-Tiered Web Applications

In this Chapter, we present Queueing Model Based Adaptive Control, which can be considered as an instance within the Queueing Model Based Feedback Control framework. It combines queueing model predictor and adaptive feedback control. We show by using adaptive control as the feedback loop, we can achieve better QoS tracking of the underlying applications compared with those schemes when non-adaptive feedback loop was used, especially in the case when the queueing model used is not very accurate. In this case, adaptive feedback controller can self-tune the control parameters in adaptation to the possible model inaccuracies and measurement errors. We evaluate the effectiveness of the proposed scheme in controlling the timing behavior via admission control for a multi-tiered Web service application.

The purpose of this chapter is two-fold. First it shows that by using different combinations of queueing models and feedback control techniques, we can get a family of Queueing Model Based Feedback Control schemes suited for different applications and different QoS goals. Secondly, we show Queueing Model Based Feedback Control has wide application to autonomic performance management. It can be applied not only in single-node server systems (as shown in Chapter 3), but also in general computing systems including multi-tiered Web applications [1].

[1] Our related publications contribute to this chapter are [60, 61]

4.1 Introduction and Motivation

In Chapter 3, we present Queueing Model Based Feedback Control, a new framework for controlling the performance of computing systems. It utilizes the modeling (descriptive) power of queueing theory and the dynamic management (prescriptive) power of feedback control, hence it can give better QoS tracking than other previously proposed approaches.

When controlling complex systems such as multi-tiered Web service applications [44, 60], we may use a simple queueing model to design the feed forward Queueing Predictor. This is because a simple model is usually easier to get, and it renders a smaller parameter space thus easier for online estimation and control purposes. On the other hand, a simpler model also means larger model inaccuracies. A simple model may not be able to capture the full dynamics of the underlying computing system. In this case, if we use a P or PI controller based on linearization of the queueing model, the performance of the Queueing Model Based Feedback control may not be fully satisfiable. This is because the queueing model applied is only an approximation to the original system, the so obtained linearized residual error model may be off from the system residual dynamics. Hence the performance of the system may degrade.

To solve this problem, in this chapter we present Queueing Model Based Adaptive Control, which uses adaptive control techniques to design the feedback control loop within the Queueing Model Based Feedback Control framework. Adaptive feedback controller can self-tune the control parameters in adaptation to the possible model inaccuracies and measurement errors. By using an example application of overload control for a multi-tiered Web site, we show that Queueing Model Based Adaptive Control can achieve better performance regulation for complex applications when accurate queueing models are hard to obtain.

The remainder of this chapter is organized as follows. Section 4.2 gives the background of overload control for multi-tiered Web applications. Section 4.3 provides the formal description of Queueing Model Based Adaptive Control approach. Section 4.4 describes our

experimental testbed and the implementation. Section 4.5 shows and discusses the experimental results in detail. Finally, Section 4.6 summarizes our conclusions.

4.2 Background

4.2.1 Muti-Tiered Web Application Architecture

Modern Web applications use a multi-tiered architecture to provide required services. While some Web applications use two tiers — Web server and database server — high volume sites typically add a third tier: application server to support complex business logic. As a consequence, most deployed Web applications utilize a 3-tiered architecture that is illustrated in Figure 4.1. This 3-tiered architecture provides a high level of scalability and reliability [14].

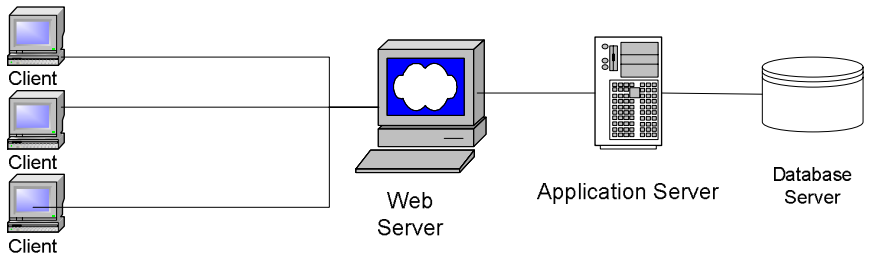


Figure 4.1: 3-Tiered Web application architecture.

In the 3-tiered architecture, on the front line of a typical Web site is the Web server that acts as the presentation layer. This tier has three functionalities: (1) receives requests from the clients and service static Web requests; (2) at the same time forwards complex dynamic content requests to the 2nd tier; (3) receives responses from the 2nd tier and sends them back to the clients. Typical Web server includes Apache and Microsoft Internet Information Server (IIS).

All the business logic for a Web site resides in the 2nd tier – application server. Application server receives requests from Web server, looks up information in the database (3rd tier) and processes the information. The processed information is then passed back to the Web server where it is formatted to be displayed on clients' machines. Typical application

servers include Apache Tomcat, Sun Java System Application Server, BEA WebLogic, IBM WebSphere, and JBOSS.

The 3rd tier – database server is the storehouse of a Web site's information. Everything from user accounts and catalogs to reports and customer orders is stored in the database. Typical database servers used in Web applications include Oracle, Microsoft SQL Server, Sybase, IBM DB2, MySQL, and PostgreSQL.

To evaluate the effectiveness of the proposed scheme in this Chapter, we built a testbed to emulate an online Web services provider in our Lab. The front-end is using Apache Web server [98], the application server is running Tomcat [99], and the backend database is running MySQL [77]. The reasons why we select this combination are:

1. All of them are open source projects and freely available;

2. Their performances are among the highest of all individual components;

3. They are widely used on the Internet, even in commercial sites such as *Amazon.com*, *Mp3.com* and *Yahoo Finance*.

Hence this combination is quite representative of the current technology. It is worth noting that the performance control approach proposed in this chapter (Section 4.3) does not depend on this specific combination and is general enough for other Web application deployments.

4.2.2 Response Time Regulation via Admission Control for an Overloaded Web Site

The QoS of a Web application is often defined as a set of criteria, referred to as Service Level Aggrements (SLAs) [37, 42]. One of the most commonly used SLAs is expressed as a maximum average response time guarantee, above which is not acceptable to the clients. In fact, in Web applications, prolonged response times usually lead to lowered usage of a site, and subsequently, reduced revenues [37].

One problem frequently encountered by Web services providers is overload [23,103], where the volume of requests for transactions at a site exceeds the site's capacity. Overload causes longer delays to the clients or even denial of service and is a major reason for SLA violations.

Admission control [24, 25] is an effective technique that prevents a system from overload. The idea is reducing the amount of work required when faced with overload by dropping a portion of the requests. This way, the server can service the accepted requests faster and meet the response time SLA. However, dropping too many requests should be prevented since this will also cause revenue loss. So the key question is, "What is the minimum portion of requests to be dropped when the Web site is overloaded in order for the accepted requests to meet their response time SLA?" An online feedback based admission control scheme is illustrated in Figure 4.2. A controller periodically takes performance measurements (measured delay[2] d) of the Web site from a monitoring agent, compares it with the desired performance (reference delay D^{ref}), and adjusts the admitting probability (P_a) to meet the performance goal (D^{ref}). The changes to the admitting probability can be actuated through an admission control (AC) module. For example, the AC module can be implemented through a proxy agent (Section 4.4). Through the AC module, a request is accepted with probability P_a, and dropped with probability $P_d = 1 - P_a$.

4.3 Queueing Model Based Adaptive Control

4.3.1 Overview of the Queueing Model Based Adaptive Control Architecture

In this section, we describe the fundamental elements of the Queueing Model Based Adaptive Control. It can be think of as an instance within the Queueing Model Based Feedback Control framework (refer to Figure 3.1). We will use the response time regulation problem

[2]In this chapter, similar to Chapter 3, we use the term "response time" and "delay" interchangeably.

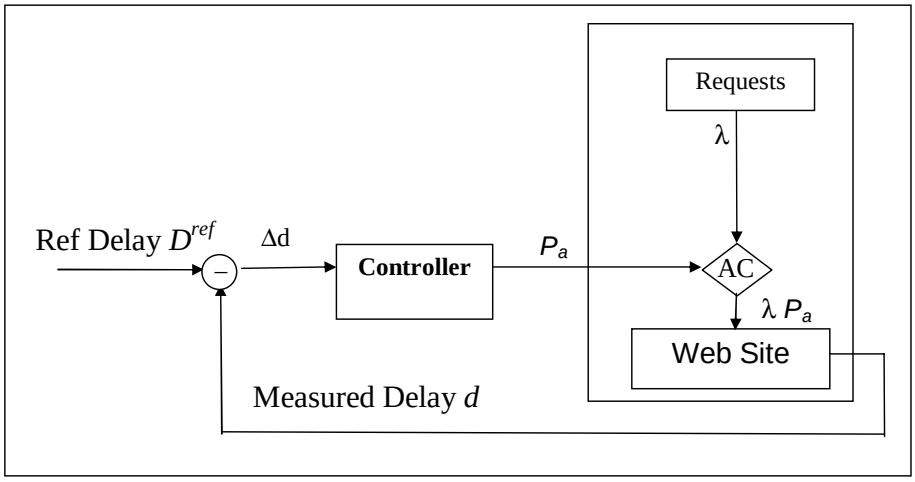


Figure 4.2: Online feedback based admission control architecture.

discussed in Section 4.2.2 as a motivating application example. Figure 4.3 shows the proposed architecture. In our proposed architecture, there is a feed forward control loop and a feedback adaptive control loop working together to output the control command (admitting probability P_a) necessary to achieve a specified average delay target (D^{ref}).

As discussed in the general architecture of Queueing Model Based Feedback Control (Section 3.2), the feed forward loop is composed of a Queueing Predictor. It takes measurements through a monitor from the computing system (Web site) to be controlled, and uses classical results from queueing theory to predict a control command (admitting probability) necessary to achieve the specified average delay target given the currently observed average workload. Let's call the admitting probability produced by this feed forward queueing predictor as P_a^m. Since queueing model used in the predictor serves only as an approximation of the real Web site, the performance of the Web site (measured average delay d) using the queueing model predicted admitting probability P_a^m may be off from the targeted delay reference D^{ref}. To correct this "residual error", we exploit an adaptive feedback loop.

The feedback control loop compares the actual delay achieved to the desired delay reference and adjusts the admitting probability accordingly in an incremental manner to ensure that the desired delay is maintained. In the discussion of Queueing Model Based Feedback

Control in Chapter 3, the controller design in the feedback loop is based on linearization of the queueing model. This is the approach used in papers [44, 94]. Here we propose to use adaptive control algorithm to design the feedback controller. This is because when the queueing model serves only as an approximation of the underlying Web site to be controlled, the further linearized model could be far off in representing the relationship between control command adjustments (ΔP_a) and the residual error corrections (Δd) well. Hence the feedback controller build on top of the linearized model may lead to degraded performances in terms of correcting the residual errors.

In Queueing Model Based Adaptive Control, we propose using adaptive control to design the feedback loop. In the adaptive control design, an online estimator will first estimate an appropriate residual error model based on the measurements of inputs (control command adjustments ΔP_a) and outputs (residual errors Δd). Then the adaptive controller will produce the adjustments of admitting probability ΔP_a based on this online estimated model. The adaptive nature of the feedback loop can help to correct errors due to model inaccuracies and disturbances coming from load changes using online measurements. Hence we anticipate the adaptive feedback loop will produce better control performance. A detailed discussion of how we design the adaptive feedback loop is given in Section 4.3.3.

As we see from Figure 4.3, the sum of queueing model predicted P_a^m and adaptive feedback loop produced ΔP_a will be used as the real admitting probability in the admission control module for the multi-tiered Web application.

4.3.2 Queueing Predictor Design

Our abstraction for the multi-tiered Web application is an M/GI/1 Processor Sharing queue (M/GI/1/PS). There are two reasons to use this simple queueing model in our design. First, modeling computing systems by a single queue is a commonly used simplification, because the performance of a computing systems is often dominated by a bottleneck stage. The M/GI/1 Processor Sharing queue abstraction encapsulates the bottleneck stage of the multi-

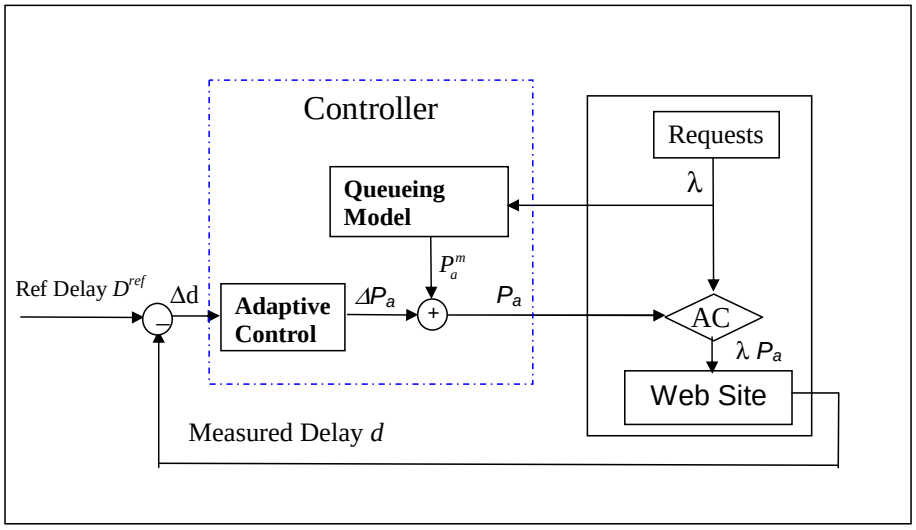


Figure 4.3: Architecture of Queueing Model Based Adaptive Control.

tiered Web application we study. Second, we want to conduct a fair comparison with other approaches in the performance evaluation (Section 4.5). Since other approaches especially the one presented in [44] also uses this simple M/GI/1/PS model, we opt using it as well. In fact, we will show even with this simple model, the performance of the resulting system is very good (Section 4.5) and out-performs previous approaches. If more accurate models such as queueing network models [60] are used, we anticipate the performance of the resulting system should be further improved.

We begin our queueing predictor design by introducing some notations similar to those presented in [44]. We denote by $RT(x)$ the mean response time of a job whose job size (or service time) is x. The job size in an M/GI/1 system is an i.i.d. random variable, denoted by X. Its probability distribution function is denoted by $F(X)$, with a mean denoted by $E(X)$. For processing sharing model [50],

$$RT_{PS}(x) = \frac{x}{1-\rho}, \tag{4.1}$$

where ρ is the load of the queue. From Equation (4.1), the mean response time for all jobs, is

$$RT_{PS} = \int_0^\infty RT_{PS}(t)\, dF(t) = \frac{E[X]}{1-\rho}. \tag{4.2}$$

Let the request arrival rate to the Web site be $\lambda(t)$. When the arrival is modulated by the admission control module with an admission probability $P_a(t)$, the effective arrival rate to the site is $\lambda_a(t) = \lambda(t) P_a(t)$. The mean response time for the admitted requests is simply:

$$RT_{PS}(t) = \frac{E[X]}{1-\lambda(t) P_a(t) E[X]}. \tag{4.3}$$

The goal of the performance control is to make the response time of admitted requests as close to the reference delay D^{ref} as possible. Suppose the queueing model is accurate, then from Equation (4.3), we know by setting

$$P_a^m(t) = \frac{D^{ref} - E[X]}{\lambda(t) \cdot D^{ref} \cdot E[X]}, \tag{4.4}$$

we can get the steady state response time to be D^{ref}. Hence Equation (4.4) gives the queueing model predicted admitting probability for the admission control.

4.3.3 Adaptive Feedback Loop Design

In this section, we present the controller design of the adaptive feedback loop. The purpose of the adaptive control loop is to correct the "residual errors" of the response time (Δd) by tuning the adjustment of admitting probability ΔP_a.

In adaptive control theory, an adaptive controller is formed by combining an online parameter estimator, which provides estimates of unknown parameters at each control instant, with a control law that is motivated from the known parameter case. The way the parameter estimator is combined with the control law gives rise to two different approaches. In the first approach, referred to as indirect adaptive control, the plant parameters are estimated

online and used to calculate the controller parameters. In the second approach, referred to as direct adaptive control, the plant model is parameterized in terms of the controller parameters that are estimated directly without intermediate calculations involving plant parameter estimates [9].

In this chapter, we apply direct adaptive control in the adaptive feedback loop design for its simplicity. In particular, we use the scheme presented in [33]. In order to construct the control law, the adaptive controller first needs to estimate a model for the system whose parameters can be used in the controller. In the following discussion, we describe the scheme using a general model:

$$A(q^{-1})y(k) = q^{-d}B_0(q^{-1})u(k), \tag{4.5}$$

where:

$$\begin{aligned} A(q^{-1}) &= 1 + a_1 q^{-1} + ... + a_n q^{-n}, \\ B_0(q^{-1}) &= b_0 + b_1 q^{-1} + ... + b_m q^{-m}, b_0 \neq 0, \end{aligned} \tag{4.6}$$

and $y(k)$ is the control output, $u\,(k)$ is the control input (command). In our admission control application, $y(k)$ corresponds to response time residual error $\Delta d\,(k)$, and $u\,(k)$ corresponds to admitting probability adjustment $\Delta P_a\,(k)$.

The model parameters of Equation (4.5) are estimated using a Recursive Least-Squares (RLS) estimator [9], which is an online version of the well-known least-square estimator.

Let

$$\begin{aligned} \phi\,(k) =& [y\,(k)\,, y\,(k-1)\,, \ldots, y\,(k-n+1)\,, u\,(k)\,, \\ & u\,(k-1)\,, \ldots, u\,(k-m-d+1)]^T, \end{aligned} \tag{4.7}$$

and

$$\begin{aligned}\theta(k) =&[\theta_1(k), \ldots, \theta_n(k), \theta_{n+1}(k), \\ &\theta_{n+2}(k), \ldots, \theta_{n+m+d}(k)]^T,\end{aligned} \tag{4.8}$$

then the RLS algorithm works as follows:

$$\begin{aligned}P(k-1) =&P(k-2) - \left[1 + \phi(k-d)^T P(k-2)\phi(k-d)\right]^{-1} \\ &\cdot P(k-2)\phi(k-d)\phi(k-d)^T P(k-2),\end{aligned} \tag{4.9}$$

$$\begin{aligned}\theta(k) =&\theta(k-1) + P(k-1)\phi(k-d)[y(k) \\ &- \phi(k-d)^T \theta(k-1)].\end{aligned} \tag{4.10}$$

Finally, the control law is given by solving:

$$\phi(k)^T \theta(k) = y^*(k+d), \tag{4.11}$$

where $y^*(k)$ is the reference input at time instance k. In our case, since we want to make the "residual errors" to diminish, we set $y^*(k) = 0$. The above algorithm begins with initial conditions $P(-1) = p_0 I$ and $p_0 > 0$.

Figure 4.4 illustrates the direct adaptive control scheme we use in the Queueing Model Based Adaptive Control architecture. It replaces the dummy adaptive control loop in Figure 4.3.

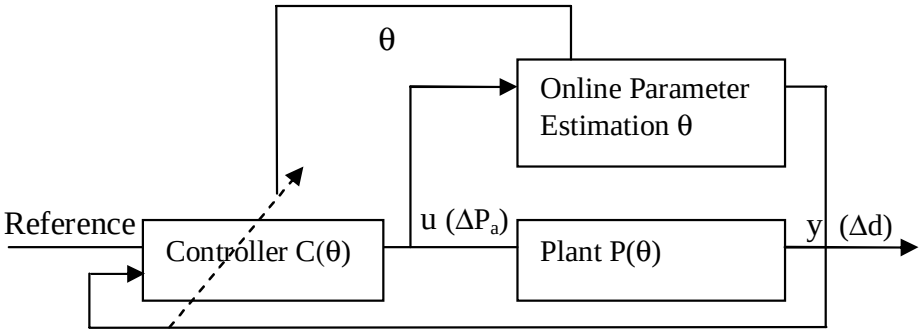


Figure 4.4: Adaptive feedback loop design using direct adaptive control.

4.4 Experiment Setup and Implementation

To validate the effectiveness of the proposed Queueing Model Based Adaptive Control, we built a testbed for response time regulation of a 3-tiered Web application using the proposed approach. In this section, we describe the testbed set-up and the hardware and software environment.

Figure 4.5 shows our testbed infrastructure. The testbed is composed of four machines. One of them is used as client workload generator and the other three machines are used as Web server, application server and database server respectively. They are connected via 100Mbps Ethernet connections. We use a proxy server for intercepting the requests and implementing the sensor, controller and actuator (Section 4.4.2). Though these modules can also be implemented in the Web server, we opt for using a separate proxy because this will make the implementation more modular and the modifications are non-intrusive to the Web application components. Non-intrusive approach is usually preferred by Web hosting companies because it can avoid possible bugs introduced by the new modules into the original components. The proxy server we use is very lightweight, so in our testbed, we put it on the same machine with the Web server to minimize communication overheads between the proxy and Web server.

The client machine is equipped with a 2.8 GHZ Intel Pentium IV processor and 512MB RAM. TPC-W client emulator [100] is used as synthetic workload generator on the client

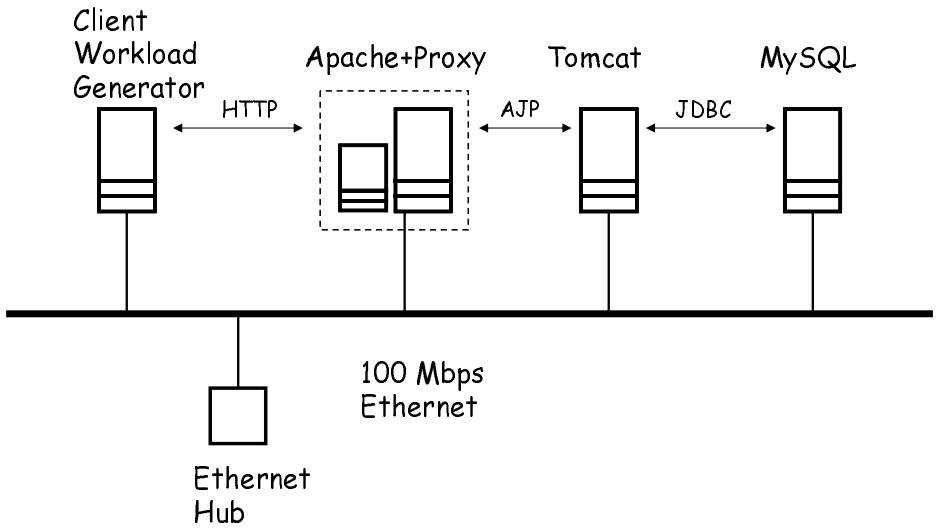


Figure 4.5: Testbed infrastructure.

(Section 4.4.1). The Apache Web server (and the proxy) machine has a 1GHZ Intel Pentium III processor and 256MB RAM, which runs Apache 2.0.7 [98]. The application server machine has a 1.5GHZ Intel Pentium III processor and 256MB RAM, which runs Tomcat 5.0 [99]. The database server machine has a 1GHZ Intel Pentium III processor and 512MB RAM, which runs MySQL 4.1.7 [77].

4.4.1 Client Workload Generator

We use TPC-W [100], an industry standard e-Commerce benchmarking tool for our E-commerce Web site testbed. TPC-W from the Transaction Processing Council (TPC) is a transactional Web benchmark specifically designed for evaluating e-commerce systems. Thus, it implements all functionalities that typical e-commerce Web sites provide, including multiple online browser sessions, dynamic Web page generations, database transactions, authentications and authorizations. TPC-W specifies 14 unique Web interactions, which are different from each other in the sense that they require different amount of server side work. Most interactions require generation of dynamic pages and database queries, range from simple select SQL statements to complicated transactions.

We modified a Java implementation of TPC-W from the PHARM group at the University of Wisconsin [15] to make it compatible with the newest version of Tomcat and MySQL installed in our testbed. It implements all functionalities in TPC-W specification. The database is configured to contain 10,000 items and 288,000 customers. In our tests, we use 12 Web interactions, except two of the original 14 defined in the TPC-W specification, since these two are only used for administrative purpose. The 12 Web interactions are rotationally generated to be sent to the Web server.

4.4.2 Controller, sensor and actuator implementation

Tinyproxy 1.6.1 [43] is modified and used to implement the sensor (monitor), controller and actuator modules. Tinyproxy is a lightweight and fast HTTP proxy that consumes less resource than fully equipped proxies such as Squid [21]. In our testbed, all HTTP requests from clients flow through Tinyproxy and are forwarded to Apache Web server. Responses from the Web server also return back to clients through Tinyproxy. In our modifications, we first implement a sensor (monitor) module. It monitors both HTTP requests from clients and responses from the Web server. The monitor also calculates parameters such as client request rates and the performance metric – average response time of requests. These parameters and metrics are then sent to the controller module via shared memory.

The controller module implements the main algorithms of Queueing Model Based Adaptive Control. It is composed of two parts: The Queueing Predictor takes the measured client request rates from the sensor module and produces the model predicted admitting probability P_a; The Adaptive Feedback Controller takes the measured average response time as input and produces the admitting probability adjustment ΔP_a. Then the combined dropping probability $P_d = 1 - (P_a + \Delta P_a)$ is sent to the actuator. In our implementation, we confine P_a, ΔP_a to be within range $(0.1, 1.0)$ and P_d to be within range $(0.0, 0.9)$.

The actuator is implemented in the following way within Tinyproxy. It randomly drops a request from clients based on the dropping probability P_d got from the controller module.

4.4.3 Determination of other parameters

The mean service time $E[X]$ to be used in the Queueing Predictor is measured offline. To this end, a very light workload is applied to the Web site. Under the light workload, since there is no queueing delay the measured mean response time approximates the mean service time $E[X]$. To remove possible measurement noises, we conducted the measurement test 5 times and averaged the measured response times to get $E[X] = 0.035$ sec.

In order to implement the adaptive controller, we need to determine the model orders d, m and n. Due to the digital implementation of the feedback controller, the effect of control command determined on time interval k can only be measured in interval $k + 1$, so we set delay order $d = 1$. For computing systems, the model orders n and m are usually fixed and low [35] and can be estimated offline. In practice, we use the following method to determine them. We first disable the adaptive feedback controller and collect offline data Δd by making use only the queueing model predictor and a white noise input of ΔP_a. Under different combinations of n and m, we use model identification method – least square estimator to find the corresponding model parameters θ, then we test which θ got from these combinations gives good fitting. By "good fitting", we mean using the θ, a new data group of collected data pairs $\{(\Delta P_a, \Delta d)\}$ produces high r^2 value [63]. In our system, we find $n = 1$ and $m = 0$ give relatively good fitting.

4.5 Experimental Results

In this section we present detailed experimental results to show the effectiveness of Queueing Model Based Adaptive Control approach. To this end, we compare the performance of four different approaches with each other. They are:

(1) Queueing model only; (feed forward queueing predictor only)

(2) Adaptive control only (approach proposed in [68] and [46]);

(3) Queueing model + PI control (approach proposed in [94] and [44], also discussed in

Chapter 3);

(4) Queueing model + Adaptive control (approach proposed in this chapter, i.e. Queueing Model Based Adaptive Control).

Among them, approach (2) is proposed in work by Lu et al. [68] and Karlsson et al. [46]. It only uses an adaptive control loop. Approach (3) is the Queueing Model Based Feedback Control using PI controller as the feedback loop design. It was published in [94] and [44]. Approach (4) is the Queueing Model Based Adaptive Control proposed in this chapter. The comparisons are based on the measurements obtained from the testbed. In all the experiments, the reference delay was set to $D^{ref} = 0.10$ second, which is a typical user acceptable response time. At each control period, the average request arrival rate and average response time of that interval are measured. These values are used in the controllers' calculations for different approaches to produce the corresponding dropping probabilities P_d. The resulting dropping probability is then set in Tinyproxy's actuator module to take into actuation.

Two sets of workloads were used in our tests. The first set, Workload A (WLA) is a simple workload which has exponentially distributed inter-arrival time with mean 0.025 sec. The second set, Workload B (WLB) is a more complicated workload which is changing. From time $t = 0$ sec to $t = 150$ sec, WLB is the same as WLA. At time $t = 150$ sec, a second workload with the same mean inter-arrival time (0.025 sec) joins in, which makes the total request rate twice as much as that of WLA.

4.5.1 Comparisons of Different Approaches under Workload A

First, we tested the four approaches under workload A. Each test runs for 300 seconds. In order for the readers to better distinguish between the result graphs of different approaches, we plot the response time measurements under workload A in two figures. Figure 4.6 compares the response time measurements of Queueing model only, Adaptive control only and Queueing model + Adaptive control. Figure 4.7 compares the response time measurements

of Queueing model + PI control versus Queueing model + Adaptive control. In these figures, x-axis is the experiment run time in seconds. Each point shows the averaged response time (y-axis) of all the requests during the past 3 seconds. Experimental results reveal the following observations:

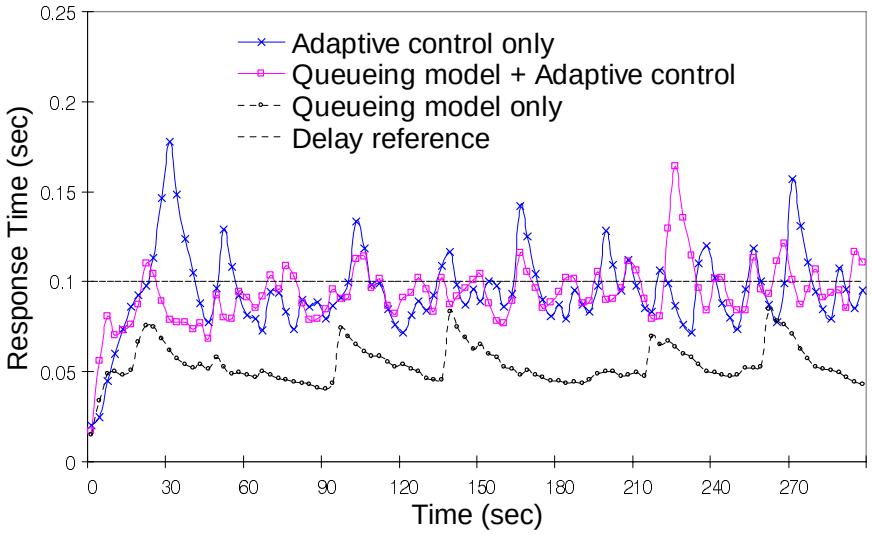


Figure 4.6: Performance comparison of Queueing model only, Adaptive control only and Queueing model + Adaptive control approaches under workload A.

Table 4.1: The aggregate errors under workload A

Workload A	Queueing model only	Adaptive control only	Queueing model + PI control	Queueing model + Adaptive control
Aggregate error	0.474	0.233	0.218	0.181

- Using the aggregate of the squared errors between the desired (reference) and actual connection delay over the duration of the experiment, we can compare the performance of different schemes. The smaller the aggregate error, the better the convergence. Table 4.1 summarizes the results.

- Using Queueing model only approach, a large steady state error develops (Figure 4.6). This is attributed to the fact that the multi-tiered Web application is not exactly a

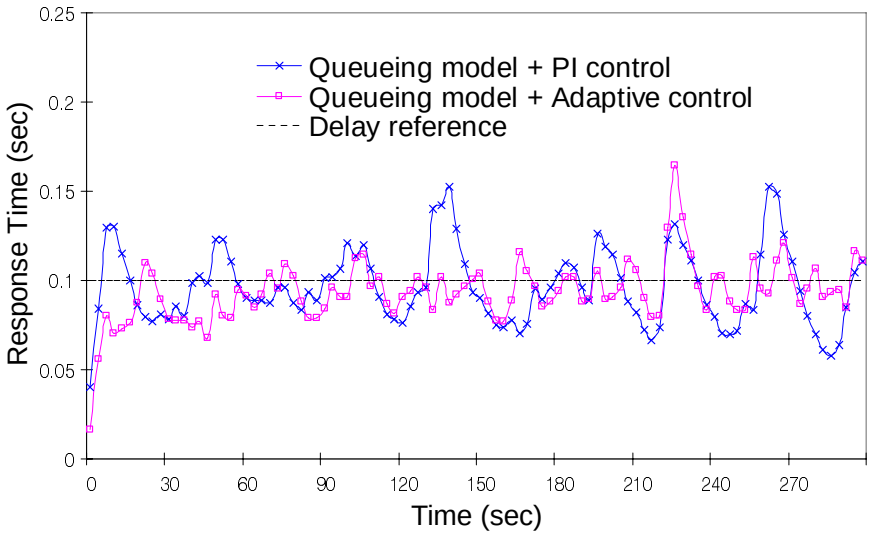


Figure 4.7: Performance comparison of Queueing model + PI control and Queueing model + Adaptive control approaches under workload A.

single-queue system. Thus, the model derived in this chapter is only an approximation of the true Web site. The fluctuation is also quite large.

- Queueing model + PI control approach's performance is better than that of Adaptive control only approach. This is attributed to the fact that Adaptive control uses linear model to approximate the real system, in which the online estimator need to run for some time before settling near the true parameter values. However, in the case of Queueing model + PI control approach, the queueing predictor is an approximation of the Web application being controlled, it can help jump to the vicinity of the true control value faster. And the PI controller will help to reduce the residual errors.

- The proposed Queueing Model Based Adaptive Control out-performs other approaches. This is because first, the queueing predictor is able to supply an approximate control value that achieves a response time close to the set point. Secondly, in terms of correcting the residual errors, the PI control loop is based on the linearization of the approximated queueing model. On the other hand, adaptive feedback loop can find better control value adjustments based on online measurements and adaptations.

4.5.2 Comparisons of Different Approaches under Workload B

Now we present the results of experiments under a changing workload — WLB. Since approaches using queueing predicator give better results, we will only show the comparison of Queueing model + PI control (approach (3)) and Queueing model + Adaptive control (approach (4)). Figure 4.8 compares the response time measurements of these two approaches. Table 4.2 shows the aggregate errors under workload B.

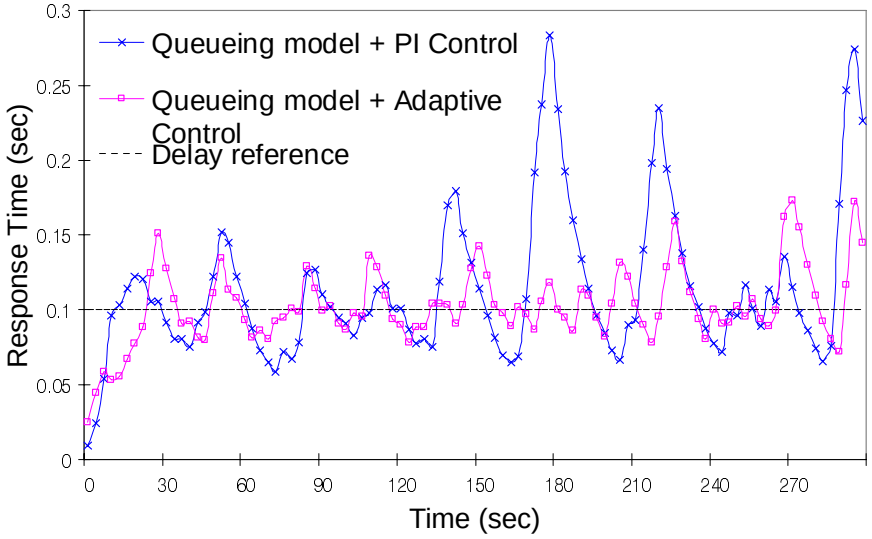


Figure 4.8: Performance comparison of Queueing model + PI control and Queueing model + Adaptive control approaches under workload B.

Table 4.2: The aggregate errors under workload B

Workload B	Queueing model + PI control	Queueing model + Adaptive control
Aggregate error	0.521	0.252

As we can see from both Figure 4.8 and Table 4.2, the proposed Queueing Model Based Adaptive Control approach has smaller error and achieves better response time regulation.

Our experimental evaluation demonstrates the advantages of integrating a queueing predictor with an adaptive feedback loop to achieve performance guarantees in Web applications.

The two components have complementary strengths, jointly offering more robust tracking of performance set points in the presence of widely unpredictable load.

While queueing theory and adaptive control theory have been repeatedly used in isolation in many contexts to provide performance guarantees, we are not aware of any prior work that demonstrates and advocates their combined use within a single framework. We believe that by exploiting different queueing models for queueing predictor design and different control techniques for feedback loop design, the framework of Queueing Model Based Feedback Control can be tailored for different QoS control goals in different applications.

4.6 Conclusions

In this chapter, we studied Queueing Model Based Adaptive Control approach for performance regulation of computing systems. In this approach, the queueing predictor and adaptive feedback controller operate concurrently in a complementary manner to make the performance of the system meet the desired target. The feed forward queueing predictor controls the system state near an equilibrium operation point, in spite of changes in the arrival process. The adaptive controller corrects errors due to the inaccuracies in the queueing model and disturbances by using online estimation and adaptation. Queueing Model Based Adaptive Control can be regarded as one instance within the Queueing Model Based Feedback Control framework. To evaluate the efficacy of the proposed approach, we build a multi-tiered Web application testbed with open-source components widely used in industry. We implemented Queueing Model Based Adaptive Control in the testbed. Experimental studies show it achieves better response time regulation than previous proposals.

Chapter 5

On-demand Real-Time Guard: A New Software Fault Tolerance Architecture

In this chapter, we study how to use feedback control to provide fault tolerance for real-time control systems. We propose ORTGA (On-demand Real-Time GuArd), a new fault tolerance architecture which exploits feedback control of software execution to achieve fault tolerance. [1]

5.1 Introduction

Real-time and embedded systems are now a central part of our lives. They have already been used in a variety of different markets, including aerospace, communication systems, automobiles, healthcare, and personal electronics to name a few. Real-time and embedded systems research is regarded as one of the next IT (Information Technology) frontier [1].

A real-time system has well-defined timing constraints. Different from general types of computer systems, a real-time system is considered to function correctly only if it returns the correct result within the system-wide timing constraints [58]. A typical definition of real-time system is "a computer system in which the correctness of the system depends not only on the logical result of computation, but also on the time at which the result is produced." [96]

An embedded system is a special-purpose computer system, which is completely encapsulated by the device it operates. Typical embedded systems include MP3 player, cellular phone, printer, ATM machine, avionic, PDA, household appliances, ABS, and medical

[1] Our related publications contribute to this chapter are [22, 59].

equipment. Since a significant amount of embedded systems are for control/communication applications, they usually have stringent temporal requirements, they are also real-time systems.

Reliable functioning of real-time systems is of paramount concern to the millions of users that depend on these systems everyday. However, fault and failures can occur in real-time systems. Failures can be caused by hardware (e.g., electro-mechanical device) malfunctions and/or faults, software (e.g., the processes/threads running on a computer) faults, or the communication channel faults between the system components for networked real-time embedded systems.

Hardware faults and communication channel faults, which are typically tolerated by hardware redundancy [26] and techniques such as message buffering [34], are not the focus of this dissertation. We will focus on how to tolerate software faults in real-time control systems using a feedback based approach.

In real-time control systems, software faults can be categorized along three dimensions [92]:

1. Resource sharing faults: corrupting other module's code and data;

2. Timing faults: failure to meet timing constraints;

3. Semantic faults: producing wrong values.

Simplex [89,92] is a software architecture which facilitates the building of dependable real-time control systems. It provides dynamic toleration of software faults. In Simplex, resource sharing faults are handled by address space protections and the protection of shared critical resource such as I/O channels through OS support. Timing faults are handled through the real-time scheduling methods such as the Generalized Rate-Monotonic Scheduling (GRMS) theory [49, 95]. Semantic faults are handled through *Analytical Redundancy* by running redundant high-assurance controller to guard the system. Simplex has been successfully

used in applications such as automated maneuvering of aircrafts [88] and semiconductor wafer-making facility [89] etc.

However, there are two drawbacks of the Simplex fault tolerance architecture:

1. In Simplex, the analytical redundant high-assurance controller runs in parallel with high-performance controller even when there is no fault occurs. This unnecessarily lowers the total CPU utilizations available to other active tasks. In most well-tested industrial applications, failures are infrequent. A parallelly-running high-assurance controller nearly doubles the CPU execution time for a single plant, makes Simplex a high-cost scheme. Since real-time embedded systems are usually resource-constraint, this drawback keeps the application of Simplex from those applications where both an efficient resource utilization and a high fault coverage are desired.

2. The design of Simplex is based on the assumption that high-assurance controller and high-performance controller are running at the same period. This assumption simplifies the scheduling analysis of real-time control tasks. However, for digital control applications, sampling/control periods[2] have performance effects on the system being controlled [9, 90, 93]. In practical applications, different controllers can be implemented at different periods for different performance considerations. Hence the original Simplex lacks the flexibility to allow different sampling implementations of the high-assurance controller and the high-performance controller. When fault occurs in using the high-performance controller, ideally the system designer prefers to run the high-assurance controller at a faster rate to help recover from the fault and protect the system promptly. So it is desirable to remove the restriction in Simplex that high-assurance controller and high-performance controller must run at the same period.

[2]In this chapter, we use the terms of a controller's "sampling/control period", "sampling period" and "control period" interchangeably. When there is no ambiguity in the context, we simply use "period" for brevity.

To tackle these drawbacks, we designed a new fault tolerance architecture called ORTGA (On-demand Real-Time GuArd). ORTGA delivers the same functionalities as the original Simplex, with the same high fault coverage and reliability. Unlike Simplex, ORTGA has advantages including that it allows more efficient resource utilization and more flexibility. In ORTGA, the high-assurance controller is running in an on-demand fashion. Only when a fault is detected, will the high-assurance controller be kicked in to replace the faulty high-performance controller. In ORTGA, at any time, only one controller will be active to control the plant. As a result, much resource is saved. ORTGA also allows the high-assurance controller and high-performance controller to be running at different periods, which gives more flexibility for control and fault tolerance design.

ORTGA exploits the idea of feedback control of software execution to achieve fault tolerance. We will discuss in detail the design principle and architecture of ORTGA in Section 5.3.

The design and implementation of ORTGA raises an important research issue: *the co-design of fault tolerance and scheduling in real-time control systems*. Two main challenges exist with respect to this problem. One is the determination of the control loop period of the high-assurance controller during a recovery. The other is the schedulability analysis of mode-changes incurred by the recovery.

With respect to the first challenge, from the fault-tolerance point of view, a larger maximum stability region allows more admissible states of the high-performance controller, hence prevents false alarms. This favors a shorter control loop period of the digital implementation of the high-assurance controller. On the other hand, from the scheduling point of view, a too short control loop period of the digital controller may lead to low schedulability of the whole task set due to its increased utilization. How to tradeoff between the fault coverage and schedulability is the main problem.

With respect to the second challenge, when the control mode is changed from the faulty high-performance controller (HPC) to the high-assurance controller (HAC) during a recovery, the execution time and control loop period may be changed too. This potentially causes

a mode-change problem, which requires detailed schedulability analysis. Further, schedulability analysis of the task set with mode-change should be considered together with the fault coverage design, as we will discuss in Section 5.5.

When there are multiple plants, there may be random multiple faults occurring, leading to possible concurrent recoveries. In this case, the schedulability analysis becomes extremely difficult. A practical and feasible solution is desired. Though the optimal co-design of fault tolerance and scheduling for multiple plants is an open problem, we provide a practical solution in this chapter.

The rest of this chapter is organized as follows. We first discuss feedback control of software execution in Section 5.2. Then we present the proposed ORTGA architecture in Section 5.3. We discuss the implementation and evaluation of ORTGA based on an inverted pendulum prototype in Section 5.4. Finally, we propose the fault tolerance and scheduling co-design problem and present its solutions in Sections 5.5 and 5.6.

5.2 Feedback Control of Software Execution for Software Fault Tolerance

Feedback is a universal mechanism which exists in many disciplines. Human uses feedback to correct faults and progress. Government uses feedback to avoid corruption and advance. Car cruise control uses feedback control to meet the targeted speed. Feedback is also commonly used in software industry: many Web sites (such as Amazon.com) use client feedbacks to improve their design; software vendors often employ user feedbacks to help select new features to be included in future releases; Microsoft uses application crash report (a kind of feedback) to improve the reliability of Windows operating system. In this chapter, we discuss using feedback to achieve software fault tolerance. Specifically, we propose ORTGA, a new fault tolerance architecture for real-time control systems.

One of the most important aspects of ORTGA is it uses *feedback control of software*

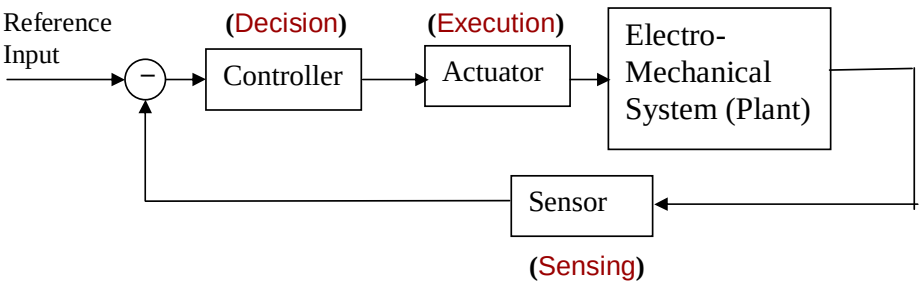


Figure 5.1: A typical feedback control loop.

execution to achieve fault tolerance. In order to understand this notion, let's first look at what are essential elements related to feedback control to make a general system (such as a social system – a government) fault tolerant?

Faults abound in any complex system such as a human government. Some kind of faults can not be easily eliminated. These include human operation errors and corruptions, to name a few. Instead, these faults/errors are facts to be coped with. The idea of fault tolerance is to respond gracefully to these faults and not make them affect the healthy operations of the whole system. The first step in achieving a fault tolerant system is to detect the faults. Common ways to detect faults in a government include auditing or collecting employee feedbacks. We call this step as *(fault) identification step*. After a fault is identified, we need to decide what is a good way to get rid of the fault, or at least confine the fault from propagating to other functional units. The result is a correction scheme. We call this second step as *decision step*. The last step is to make sure the correction scheme is executed in order to correct the fault occurred. We call the last step as *execution step*.

These three steps correspond to a typical feedback control loop for an electro-mechanical system, as shown in Figure 5.1. The (fault) identification step is similar to sensing, where the sensor finds state or output errors and feed them back to the controller. The decision step is similar to control, where the controller calculates the control values to correct the error. The execution step is similar to the actuation, where the actuator puts the control values from the controller into action.

Following this analogy, we now introduce the architecture of ORTGA. We also discuss

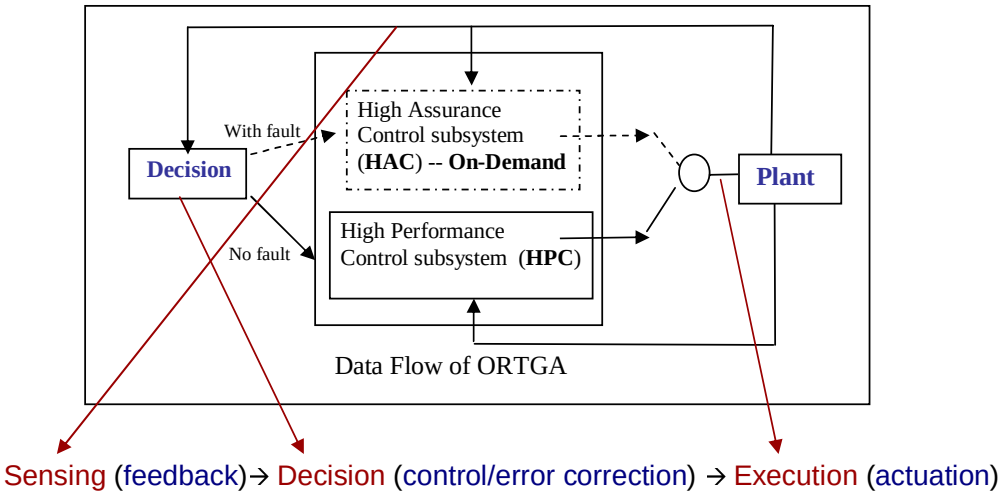


Figure 5.2: Feedback Control of Software Execution On Demand.

how ORTGA employs feedback to make a real-time control system fault tolerant. The architecture of ORTGA is shown in Figure 5.2. Similar to the Simplex architecture [89], in ORTGA the software component of the plant under protection is divided into a high-assurance-control (HAC) subsystem and a high-performance-control (HPC) subsystem.

The HAC subsystem is a control software which was proved to be reliable. HAC's simple construction let the system designer leverage the power of formal methods and a rigorous development process. From the system level, high-assurance OS kernels such as certifiable runtimes are usually used in the HAC. From the application level, well-understood classical controllers designed to maximize the controlled plant's stability region is also used.

The HPC subsystem complements the conservative HAC core. From the application level, an HPC can use more complex and advanced control technologies for higher control performance. These can include complex and advanced technologies that are difficult to verify, for example, neural network control. From system level, COTS real-time OS and middleware designed to simplify the application development can be used in HPC. However, these software components may not be certifiable and could contain faults.

Unlike Simplex, in ORTGA the HAC and HPC subsystems do not run in parallel. At any time, there is *only one* instance of either HAC or HPC is running. Normally, the HPC

controls the plant. However, the decision logic in the decision module ensures that the plant state under the HPC stays within an HAC-established stability region [91][3]. If this is violated, the HAC will be kicked in and takes over the system. In the original Simplex architecture, it requires running the two controllers — HAC and HPC — for each plant in parallel. This "trade system resource for safety" approach makes the whole system inefficient, hence limits the application of Simplex only to those extremely safety-critical applications where cost and resource usage is not a major concern.

Furthermore, in ORTGA, the HAC and HPC can be running at different periods. This allows for flexibility in designing the HAC and HPC. For example, when HPC is detected faulty, the carefully designed HAC can run at a faster rate to help recover the plant promptly.

As we can see from Figure 5.2, ORTGA achieves fault tolerance by using *feedback control of software execution on demand*. At every decision time, the decision module gets the state feedback from the plant and determines if the current state is still within the HAC-established stability region. If it is, the HPC still controls the plant; otherwise, the HAC is activated and it takes over the control of the plant. The decision module determines which output should be used for the plant. Then the plant will execute the control output values accordingly. These *Sensing(feedback)* $\rightarrow$ *Decision(control)* $\rightarrow$ *Execution(actuation)* steps constitute the feedback control of software execution (cf. Figure 5.1). By using the HAC to guard against possible faults in the HPC in real-time, ORTGA achieves fault tolerance.

5.3 ORTGA Design

In this section, we present the architecture and design considerations of ORTGA.

[3]A method to determine stability regions for digital controllers is presented in Section 5.5.1

5.3.1 Components Organization and Fault Recovery Procedure of ORTGA

We call a plant for which ORTGA provides fault tolerance protection a *(ORTGA)* **FT-enabled plant**. In ORTGA, there can be multiple FT-enabled plants. For each FT-enabled plant, similar to Simplex, there are 3 logical modules within ORTGA: 1) a decision module; 2) a high-performance controller (HPC); and 3) a high-assurance controller (HAC).

The decision module plays a key role in providing fault detection and recovery using its decision logic, which ensures that the state of a plant under the HPC's control always stays within the HAC-established maximum stability region, i.e. $x^T Px < 1$. [4] If HPC's control command is expected to make the plant state violate this condition, the HAC takes over. In practice, we use a smaller stability region, for example, $x^T Px < 0.8$. The distance between $x^T PX = 1$ and $x^T Px = 0.8$ is the margin reserved to guard against errors in the model, and inaccuracies in the sensing and actuation. The decision module also guards against HPC's faults in the temporal dimension, such as budget overrun, deadline miss.

ORTGA runs the HAC only when it is necessary, i.e. only when a fault/failure is detected in the HPC by the decision module. By eliminating redundant executions of the HAC, the run-time overhead of ORTGA is significantly lower than that of Simplex. The saved CPU cycles can be used by other real-time tasks sharing the same CPU.

The elimination of redundant execution of controllers is realized in the following way. On system start-up, all components are started but the HAC is blocked. As soon as a fault is detected in the HPC, the decision module suspends the HPC and activate the HAC. Now the plant is under the control of HAC for recovery. If the type of fault is semantic (e.g. control command out of stability region), the HPC is allowed to be switched back later, after the HAC stabilizes the plant. If the failure is due to an execution error such as segmentation fault or infinite loop, the HPC will be restarted for later retry.

[4] The definition and procedure to calculate P is discussed in Section 5.5.1.

When the plant has been stabilized under the HAC, the control will be switched back from the HAC to the HPC for retry or performance considerations. Unlike the recovery, which must be done in a timely fashion in order for the plant state to stay within the HAC-established maximum stability region, the initiation time of the switch-back can be more flexible. For example, in practice, the decision module can initiate the switch-back when the plant state is near the control set point and the CPU is in an idle interval (to avoid possible mode-change problems which will be discussed in Section 5.5).

5.3.2 CPU Usage Savings Analysis of ORTGA

In this section, we discuss the resource savings of ORTGA quantitatively in terms of CPU usage.

We use the common adopted periodic task model to model the control tasks. Each periodic task is denoted by τ_i. The *timing parameters* of a task τ_i is represented in the tuple (C_i, T_i), where C_i is the worst-case execution time, and T_i is the task period. For control systems, the task deadline is usually the same as its period, i.e., we have $D_i = T_i$.

Compared with Simplex, the CPU resource usage of ORTGA is greatly reduced due to the on-demand execution of HAC. Suppose the timing parameters of an FT-enabled plant under the control of its HPC is (C^p, T^p), while the timing parameters of the same plant under the control of its HAC is (C^a, T^a). We use P_r to denote the percentage of time used for recovery (i.e., when HAC is active) during a total of run time of T (in the unit of milliseconds).

In original Simplex, the total CPU resource usage (in the unit of milli-seconds) is roughly:

$$R_{Simplex} = (1 - P_r) \cdot \left(C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil + C^p \cdot \left\lceil \frac{T}{T^p} \right\rceil \right) + P_r \cdot C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil . \tag{5.1}$$

While in the new design of ORTGA, the total CPU resource usage (in the unit of mil-

liseconds) is roughly:

$$R_{ORTGA} = (1 - P_r) \cdot C^p \cdot \left\lceil \frac{T}{T^p} \right\rceil + P_r \cdot C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil \quad . \tag{5.2}$$

Compared with Simplex, the new design saves $(1-P_r)\cdot C^a \cdot \left\lceil \frac{T}{T^a} \right\rceil$ in terms of CPU resource usage. In practical applications where faults are infrequent, P_r is very small, thus ORTGA saves much of the CPU resource.

5.3.3 Extra One Period Delay of ORTGA

We see the on-demand running of HAC in ORTGA saves much of the CPU resource, however, there is up to one period delay in the recovery due to the on-demand recovery. Figure 5.3 illustrates the extra delay incurred during the recovery using ORTGA. Suppose at time t, a fault is detected in the HPC. The upper half figure shows the recovery timeline of the original Simplex, while the lower half figure shows the recovery timeline of ORTGA. For Simplex, since HAC is running in parallel with HPC, when fault is detected, the control output of HAC can be applied at the beginning of the next control period (t^1) since it is already computed in the current period. For ORTGA, HAC is running on-demand. When fault is detected, HAC need to be kicked in and the control output be computed during the first period of HAC's running. So the control output of HAC can only be applied at one period (T_k^a) later, i.e. the 1st time HAC's control takes action is at time t^2. As a result, compared with the recovery using Simplex, the recovery using ORTGA will incur an extra delay up to $t^2 - t^1 = T_k^a$.

The extra delay can be compensated by using state projections. The idea is illustrated in Figure 5.4. At any decision time t, the decision module project the plant state for one period of HAC more, i.e. $x(t + T_k^a)$. If the projected state is still in the recovery region associated with the HAC, then HPC can still be running; otherwise, HAC is kicked in to take over.

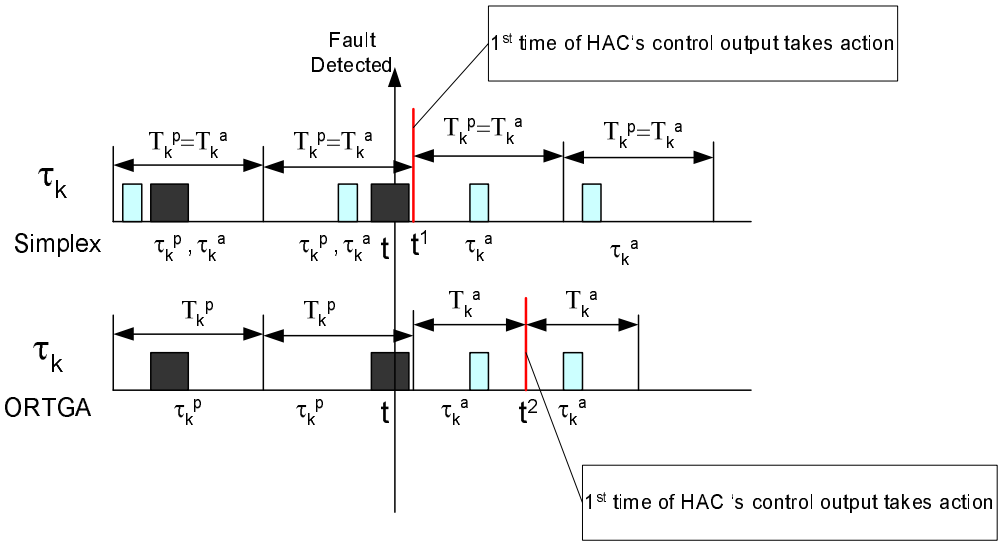


Figure 5.3: Illustration of the extra delay in recovery using ORTGA.

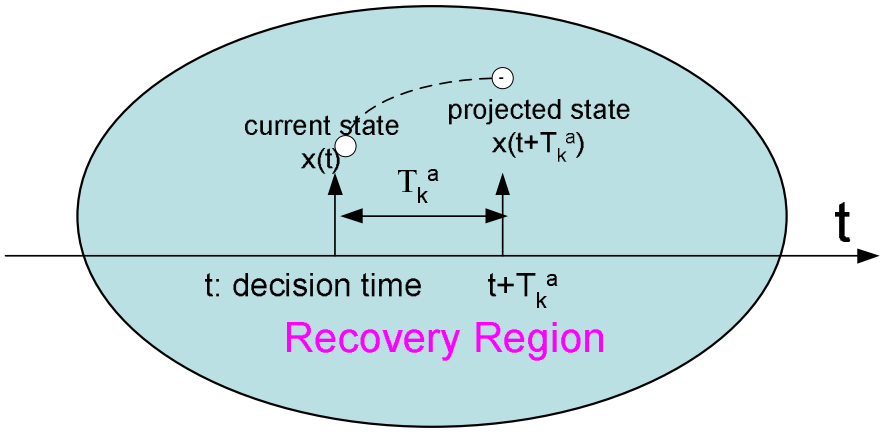


Figure 5.4: Illustration of the use of state projection to handle the extra delay.

Since in ORTGA, the period of HAC (T_k^a) can be smaller than that of HPC (T_k^p), the extra delay's impact can be small compared with the increased recovery region of the HAC due to ORTGA's flexibility which will be discussed in Section 5.5.2.

A simple and computationally-inexpensive state projection method is presented in Section 5.5.1.

5.4 Implementation and Evaluation of ORTGA

We have implemented ORTGA on an inverted pendulum testbed. Figure 5.5 shows the inverted pendulum control system which demonstrates the effectiveness of ORTGA. This inverted pendulum is inherently instable [53]. When there are several missed control outputs, the inverted pendulum would fall even when the angle error and angle velocity are small (with respect to the erected position). There is a strict timing requirement for this inverted pendulum system to maintain stability. This is because if it tilts past a certain angle, even one missing control outputs from the controller would lead the pendulum to fall. As a result, the fault detection and recovery must be carried out in a timely and predictable manner [53]. The control computer which runs ORTGA is based on a Pentium II 350MHz processor with 66MHz memory bus. It has 32KB of level 1 cache memory on chip. Quadrature encoder interface is used for sensing input and a digital to analog converter is used for control output. Our implementation of ORTGA is written in C [47]. It runs on Linux kernel version 2.4.18-3 with RT scheduling and kernel preemption enhancements. ORTGA uses a fixed priority scheduling scheme — Rate Monotonic Scheduling [57], as it is widely supported by current systems and standards such as the POSIX real-time extension [87].

In the following, we discuss the evaluation of ORTGA.

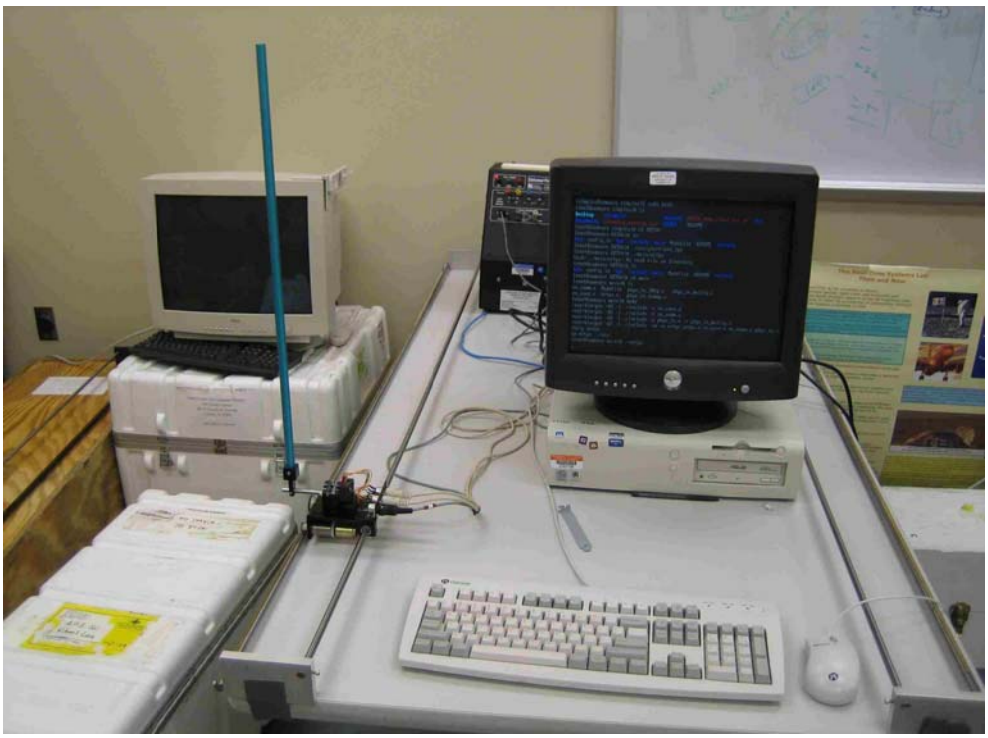


Figure 5.5: Inverted pendulum control system protected by ORTGA.

5.4.1 CPU Resource Savings of ORTGA

In order to measure the execution time savings of ORTGA compared with the original Simplex, we collected the controllers' running temporal data from the prototype testbed we built.

We measure the HPC and the HAC controller's execution times by using the *rdtscl*() call [83]. It reads the lower half of the 64 bit hardware counter RDTSC (Read Time Stamp Counter). RDTSC is provided on Intel Pentium-family processors.

rdtscl() macro is defined as follows:

```
#define rdtscl(low) \
    __asm__ __volatile__("rdtsc" : "=a" (low) : : "edx")
```

We get the RDTSC readings before a controller is executing and after the controller is executing. The difference gives the clock cycles used by the controller's execution. The

controller's execution time equals to the clock cycles used divided by the clock speed. On Linux, the actual clock speed can be found in $/proc/cpuinfo$. For our testbed, the CPU clock speed is 350MHz.

We collected multiple run samples for both the HPC and the HAC with sample sizes larger than 500. Table 5.1 shows the the mean, variance, minimum, and maximum of execution times of the HPC and the HAC respectively.

Table 5.1: Execution statistics for HPC and HAC

Controller	Average Execution Time (μs)	Variance of Execution Time	Minimum Execution Time (μs)	Maximum Execution Time (μs)
HPC	2.6705	0.02181	2.3571	3.2857
HAC	1.1060	0.005812	0.9429	1.6371

As we can see from Table 5.1, if the HPC and the HAC are running at the same periods, the percentage of CPU usage savings (in terms of controllers' execution times) using ORTGA compared with that of using Simplex is (cf. Equations (5.1) and (5.2))

$$\frac{(1-P_r)\cdot 1.1060}{(1-P_r)\cdot(2.6705+1.1060)+P_r\cdot(1.1060)} \cong \frac{1.1060}{2.6705+1.1060} = 29.29\%.$$

where P_r is the percentage of time used for recovery, which is assumed to be small.

In ORTGA, the HPC and the HAC can be run at different periods. For example, in our tests, the HPC is running at 20Hz and the HAC is running at 50Hz. Consider the different invocation frequencies of the HPC and the HAC, the percentage of CPU usage savings (in terms of controllers' execution times) using ORTGA compare with that of using Simplex is:

$$\begin{aligned}&\frac{(1-P_r)\cdot 1.1060\cdot 50}{(1-P_r)\cdot(2.6705\cdot 20+1.1060\cdot 50)+P_r\cdot(1.1060\cdot 50)}\\ \cong&\frac{1.1060\cdot 50}{2.6705\cdot 20+1.1060\cdot 50} = 50.87\%.\end{aligned}$$

5.4.2 Fault Tolerance under ORTGA

In order to test the fault tolerance capabilities of ORTGA, we conducted extensive tests on the ORTGA testbed we built. The testing procedure is that we insert various faults/bugs into the HPC controller, and see if ORTGA can tolerate these faults and make the inverted pendulum stable.

Below we list a subset of the faults/bugs we have tested against ORTGA. We could not include all the tests we have performed due to the space limit of this dissertation. Also, this list is not intended to be complete, rather it is intended to give an overall feel of the broad range of faults ORTGA can tolerate. For all the bugs tested, ORTGA can tolerate the bug. When the system under the control of HPC failed the decision module's safety test, ORTGA replaces the HPC with the HAC. Hence the inverted pendulum can keep running without falling down.

For each bug we tested, we include in here a pseudo-code snippet to produce that bug. In the pseduo-code snippet, *glob_ctr* is a non-negative integer which represents the global counter of how many control loops have been performed by the HPC. Function *normal_HPC_cmd*() is the normal output calculation of the HPC without bugs. Function *cal_HPC_cmd*() is the HPC control output calculation function which is used to insert bugs into the HPC.

1. Infinite loop bug

The (faulty) HPC controller performs normal operation for a while, say 20 seconds (or after 400 control loops when the control rate is 20Hz), then it goes into an infinite loop.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
    float hpc_volts;
    hpc_volts = normal_HPC_cmd(...);
    if (glob_ctr++ > 400){
```

```
            while (1) {
            };
        }
        return hpc_volts;
    }
```

2. Non performing bug

The (faulty) HPC controller performs normal operation for a while, then it outputs control command of 0. In the inverted pendulum system, this corresponds to outputting a control voltage of 0 volt.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
    float hpc_volts;
    hpc_volts = normal_HPC_cmd(...);
    if (glob_ctr++ > 400)
        hpc_volts = 0.0;
    return hpc_volts;
}
```

3. Maximum control output bug

The (faulty) HPC controller performs normal operation for a while, then it outputs the maximum control command allowed by the actuator. In the inverted pendulum system, this corresponds to outputting the control voltage of 5 volts.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
    float hpc_volts;
    hpc_volts = normal_HPC_cmd(...);
```

```
    if (glob_ctr++ > 400)
        hpc_volts = 5.0;
    return hpc_volts;
}
```

4. Bang-bang type bug

The (faulty) HPC controller performs normal operation for a while, then it outputs the maximum and minimum control values in a bang-bang manner. In the inverted pendulum system, this corresponds to outputting control voltages of $+5$ volt and -5 volts in every other period.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
    float hpc_volts;
    hpc_volts = normal_HPC_cmd(...);
    if (glob_ctr++ > 400){
        if ((glob_ctr%2)==0)
            hpc_volts = 5.0;
        else
            hpc_volts = -5.0;
    }
    return hpc_volts;
}
```

5. Positive feedback control bug

The (faulty) HPC controller performs normal operation for a while, then it outputs the positive feedback control values. The positive feedback will make the controlled system unstable.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
    float hpc_volts;
    hpc_volts = normal_HPC_cmd(...);

    if (glob_ctr++ > 400)
        hpc_volts = -hpc_volts;

    return hpc_volts;
}
```

6. Divided by zero bug

The (faulty) HPC controller performs normal operation for a while, then it calculates the control value with a faculty divided by zero operation.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
    float hpc_volts=0.0;
    hpc_volts = normal_HPC_cmd(...);
    if (glob_ctr++ > 400)
        hpc_volts = 1.0/0;

    return hpc_volts;
}
```

7. A tricky designer bug

This bug is designed by Dr. Walter Heimerdinger from Honeywell ACS Laboratories. It is a more tricky and sophisticated attack to test the fault tolerance capabilities of ORTGA.

The code to insert this bug is:

```
float calc_HPC_cmd(...) {
```

```
        float hpc_volts=0.0;
        static float wdither = 0.0;
        static int wratio = 1;
        static float wincremental = 0.30;
        hpc_volts = normal_HPC_cmd(...);

        glob_ctr++;
        if ((glob_ctr % 1000) == 0) {
             wincremental += 0.80;
             wdither = wincremental;
         }

        if ((glob_ctr % 100) == 0) {
             wratio += 10;
        }
        if ((glob_ctr % wratio) == 0) {
             if (wdither == wincremental)
                 wdither = -wincremental;
             else if (wdither == -wincremental)
                 wdither = wincremental;
        }

        hpc_volts += wdither;

        return hpc_volts;
}
```

5.5 Fault Tolerance and Scheduling Co-Design – Single FT-enabled Task Case

The design of ORTGA raises an interesting research issue: *co-design of fault tolerance and scheduling for real-time control systems*. In this and the following sections, we discuss this problem for single FT-enabled task case and multiple FT-enabled tasks case respectively.

For a digital implemented controller, generally the smaller the sampling period, the larger its maximum stability region. However, a too small sampling period may make the whole system become unschedulable. From the fault tolerance point of view, the system designer prefers a larger maximum stability region, hence favors smaller sampling period. This contradicts to the view from system schedulability, which favors a larger sampling period. The key to fault tolerance and scheduling co-design is to find the optimal control period(s) for the HAC(s) in order to both maximize the fault coverage and at the same time meet the system schedulability constraints.

In this section, we first discuss in general the co-design problem, then we give an optimal solution for the scenario when there is only one FT-enabled task in ORTGA. In Section 5.6, we extend our results to the general case when there are multiple FT-enabled tasks.

We begin the discussion with the relationship between a digital controller's sampling period and its maximum stability region. To this end, we first give a method to derive the maximum stability region for a digitally implemented controller.

5.5.1 Maximum Stability Region for Digital Controllers

ORTGA's forward recovery scheme is based on the maximum stability region of the plant under the HAC controller. In this section, we propose an approach to determine the maximum stability region for a discretized control system.

Assume the plant to be controlled is governed by a continuous-time state space model as

below:

$$\frac{dx(t)}{dt} = Ax(t) + Bu(t), \tag{5.3}$$

where $x \in \mathcal{R}^n$ is the system state and $u \in \mathcal{R}^m$ is the control input. $A \in \mathcal{R}^{n\times n}$, $B \in \mathcal{R}^{n\times m}$ are the corresponding system matrices. Controllers are typically designed in a state feedback form, i.e. $u(t) = -Kx(t)$, where K is the corresponding controller gain.

Modern control systems are typically implemented on digital computers. Due to the digital implementation of continuous controllers, the sampling and control of the continuous-time system (5.3) is enforced at discrete time points. As a result, for the purpose of design and analysis, we need to convert the continuous-time system to its discrete-time form according to the digital implementation method used. For example, continuous-time controlled system (5.3) with sampling period h and a zero-order hold is represented as follow [10]:

$$x(k+1) = Fx(k) + Gu(k), \tag{5.4}$$

where $x(k) \triangleq x(kh)$ is the state of the plant at the kth sample time, and

$$\begin{aligned} F &= e^{Ah}, \\ G &= \int_0^h e^{As} ds B. \end{aligned} \tag{5.5}$$

Using Equation (5.5), we can get a simple and computationally-inexpensive state projection method to compensate the extra delay when use ORTGA as discussed in Section 5.3.3. That is, at control interval k, since we have the knowledge of the current state $x(k)$ and the current control $u(k)$ to be applied, we can get the predicted plant state at interval $k+1$ according to Equations (5.4) and (5.5).

Corresponding to the continuous-time state feedback controller $u(t) = -Kx(t)$, the digital state feedback controller is $u(k) = -Kx(k)$. By replacing the $u(k)$ term with $-Kx(k)$ in the discrete-time system equation (5.4), we get the closed-loop discretized control system

as

$$x(k+1) = \bar{F}x(k), \tag{5.6}$$

where $\bar{F} = F - GK$.

In order to determine the stability of a closed-loop discretized control system as (5.6), we use the well-known Lyapunov stability criteria which is summarized in the following theorem [10].

Theorem 5.5.1. *A discrete time linear time-invariant (LTI) system (5.6) is stable iff there exists a positive definite matrix $P > 0$ such that*

$$\bar{F}^T P \bar{F} - P < 0. \tag{5.7}$$

For a real-life application, due to the limitation on physical plant and control actuators, there are constraints on the system states and/or control inputs. For system (5.4), these constraints can be represented as:

$$a_i^T x \leq 1, \qquad i = 1 \cdots l, \tag{5.8}$$

$$b_j^T u \leq 1, \qquad j = 1 \cdots r. \tag{5.9}$$

With digital controller $u(k) = -Kx(k)$, the constraints (5.8)(5.9) can be combined and represented as a polytope (a multi-dimensional figure whose faces are hyperplanes) in the system's state space:

$$\alpha_m^T x \leq 1, \qquad m = 1, \cdots, q, \tag{5.10}$$

where $\alpha_m^T = a_m^T$, for $m = 1, \cdots, l$; $\alpha_m^T = b_j^T K$, for $m = l+1, \cdots, q$, $j = 1, \cdots r$, and $q = l + r$. The states inside the polytope are called admissible states, because they obey the operational constraints.

A stability region is defined as a subset of the states within the polytope such that if

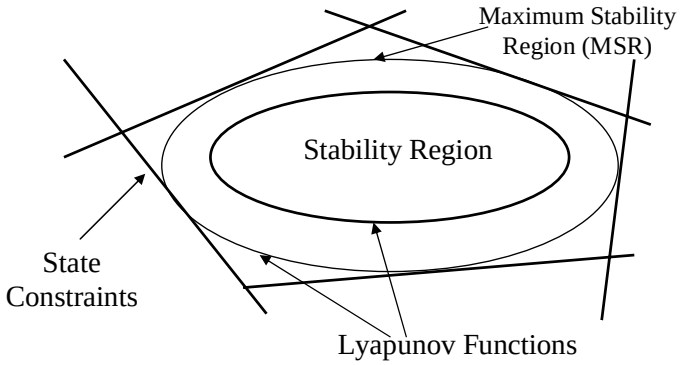


Figure 5.6: Illustration of a stability region under state constraints.

the closed-loop discretized control system starts from a state within the stability region, the system states' future trajectory will always stay within the region and finally converge to the control set point.

A Lyapunov function $x^T Px$ inside the state constraint polytope represents a stability region [88, 89]. Geometrically, it defines an n-dimensional ellipsoid in the n-dimensional system state polytope, as illustrated in Figure 5.6, where the state space is 2-dimensional. An important property of a Lyapunov function is: if the system state is within the ellipsoid associated with a controller, it will always stay within the ellipsoid and finally converge to the equilibrium position (set point) under this controller.

Mathematically, for a closed-loop discretized control system (5.6), a *stability region* under a specific Lyapunov matrix P with state constraints can be defined as the following ellipsoid:

$$S_P \triangleq \{x | x^T Px < 1\}, \tag{5.11}$$

where P satisfies the Lyapunov stability criteria $\bar{F}^T P \bar{F} - P < 0$ (Equation (5.7)) and x satisfies state constraints (Equation (5.10)).

However, a Lyapunov matrix P is not unique for a given stable closed-loop discretized control system. In order not to unduly restrict the state space within the operational constraints, we should find the maximum stability region (MSR). To get the MSR, we first give Lemma 5.5.1.

Lemma 5.5.1. *Given a discretized LTI system (5.6) with state constraints (5.10), the stability region S_P defined in (5.11) satisfies the constraints in (5.10) iff $\alpha_m^T P^{-1} \alpha_m \leq 1$, $m = 1, \cdots, q$.*

Proof. Please refer to [91] (Lemma 4.1) for a proof. □

Notice that the area of the stability region defined in (5.11) is proportional to the determinant of matrix P^{-1}. Based on Lemma 5.5.1, the determination of the MSR of a closed-loop discretized control system (5.6) with constraints (5.10) is reduced to the following Linear Matrix Inequality (LMI) problem [17].

Problem 1. *Maximize* $\log \det P^{-1}$

$$\begin{aligned} s.t.: \qquad P &> 0, \\ \bar{F}^T P \bar{F} - P &< 0, \\ \alpha_m^T P^{-1} \alpha_m &\leq 1, \quad m = 1, \cdots, q. \end{aligned}$$

Further, if $\bar{F}^{-1}$ exists and let $Q \triangleq P^{-1}$, we can convert the above LMI problem as the following new problem.

Problem 2. *Maximize* $\log \det Q$

$$\begin{aligned} s.t.: \qquad Q &> 0, \\ Q\bar{F}^T - \bar{F}^{-1} Q &< 0, \\ \alpha_m^T Q \alpha_m &\leq 1, \quad m = 1, \cdots, q. \end{aligned}$$

Problem 2 is a MAXDET problem [17]. It can be solved by using the software packages such as *sdpsol* [105] or $YALMIP$ [65]. Note that the maximum stability region and its solution via LMI formulations (i.e. Problems 1 and 2) are different from those presented in [91] for the Simplex architecture. Here we are dealing with discretized system under digital

controllers, while the authors in [91] were dealing with continuous system under continuous controllers.

The resulting $P = Q^{-1}$ from the MAXDET problem above defines the maximum stability region $\{x|x^T Px < 1\}$ in the system state polytope (see Figure 5.6).

5.5.2 Maximum Stability Region v.s. Control Period

In Section 5.5.1, we show how to derive the maximum stability region (MSR) of a system under a digital controller. When the continuous-time system model, the underlying continuous-time controller and its control loop period are given, we can get the MSR of the corresponding closed-loop discretized control system. It is obvious that the maximum stability region is a function of the control loop period T, hence we denote the MSR for a plant under a digital controller with respect to control loop period T as $MSR(T)$. We further use $A(T)$ to denote the area of $MSR(T)$. $MSR(T)$ encloses the admissible states of the system to keep it stable with respect to the control loop period T. For this reason, we call $A(T)$ the **stability index** of a specific closed-loop discretized control system.

Figure 5.7 illustrates a typical chart of stability index v.s. control loop period for a digital control system. In this example, the underlying plant is an inverted pendulum. The continuous-time system model and the high-assurance controller of the inverted pendulum are shown below:

$$\dot{x}(t) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & -2.7528 & -10.9526 & 0.0043 \\ 0 & 28.5812 & 24.9179 & -0.0441 \end{pmatrix} x(t) + \begin{pmatrix} 0 \\ 0 \\ 1.9432 \\ -4.4385 \end{pmatrix} u(t),$$

$$u(t) = -[-5.7807 - 42.2087 - 14.0953 - 8.6016]x(t) \quad . \tag{5.12}$$

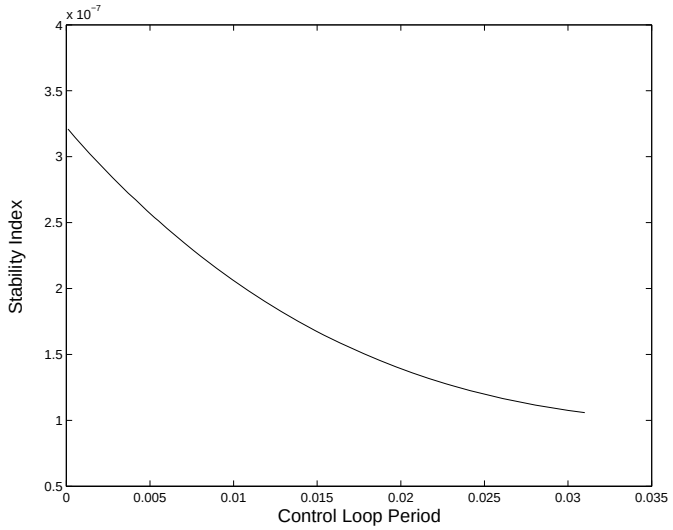


Figure 5.7: Stability index v.s. control loop period.

Here the plant state is $x = \{x_1, x_2, x_3, x_4\}$. x_1 is the inverted pendulum's (IP's) track position, x_2 is the IP's angle, x_3 is the IP's track position velocity, and x_4 is the IP's angle velocity.

The physical meaning of the decreasing shape of the stability index shown in Figure 5.7 is clear. As the control loop period decreases, the system becomes more stable, hence the stability index $A(T)$ increases. On the other direction, as the control loop period increases, the plant becomes less stable, hence the stability index $A(T)$ decreases.

5.5.3 Schedulability Analysis for ORTGA Fault Tolerance Architecture

Unlike Simplex, the new ORTGA architecture saves CPU resource which could be used for other real-time tasks. In this section, we discuss the schedulability analysis of ORTGA together with these real-time tasks.

Consider an FT-enabled plant PL under the protection of ORTGA. The task when PL is being controlled by the HPC is denoted as τ^p and the task when PL is being controlled by the HAC is denoted as τ^a. Their timing parameters are (C^p, T^p) and (C^a, T^a) respectively.

When a fault is detected in the HPC, the decision module will issue a recovery request (RR) to initiate the recovery procedure. As a result, τ^p will be aborted and the new task τ^a will be activated for recovery.

Using similar notations, the decision module is modeled as real-time task τ_d^p when PL is controlled under the HPC and modeled as task τ_d^a when PL is controlled under the HAC. In ORTGA, when controller switches, the decision module's period also switches accordingly. So tasks τ^p (controller) and τ_d^p (decision module) have the same phase and period. From a schedulability analysis perspective, τ^p and τ_d^p can be modeled as one single task $\tilde{\tau}^p$. Its execution time is $\tilde{C}^p = C^p + C_d^p$, its period is $\tilde{T}^p = T^p = T_d^p$. Similarly tasks τ^a and τ_d^a can be modeled as one single task $\tilde{\tau}^a$, where $\tilde{C}^a = C^a + C_d^a$, $\tilde{T}^a = T^a = T_d^a$.

As a result, when scheduled together with other real-time tasks, the decision modules and controllers for a single FT-enabled plant can be modeled as an abstract task $\tilde{\tau}$. It has two subtasks $\tilde{\tau}^p$ and $\tilde{\tau}^a$, where $\tilde{\tau}^p$ is running when the plant is under the control of HPC and $\tilde{\tau}^a$ is running when the plant is under the control of HAC. Task $\tilde{\tau}$ is called an *(ORTGA)* **FT-enabled task**. This model abstraction simplifies the following schedulability analysis. In the discussions below, without confusion, we use τ_k to denote the combined decision and control task for an FT-enabled plant k together with other real-time tasks.

Task Model and Definitions

We assume there are total N real-time tasks, $\tau_1, \cdots, \tau_N$, running on the CPU. They are ordered in the sequence of their priorities, with τ_1 having the highest priority and τ_N having the lowest priority. Among them, tasks τ_k, $(k \in \mathcal{FT})$ are the FT-enabled tasks, $\mathcal{FT} \subseteq \{1, \cdots, N\}$ is the set of all FT-enabled tasks. Each τ_k, $(k \in \mathcal{FT})$ is composed of two subtasks: τ_k^p represents the combined decision and control task when the plant is being controlled by the HPC; τ_k^a represents the combined decision and control task when the plant is being controlled by the HAC.

Definition 5.5.1. *Given the task set* $\{\tau_i\}$, $i \in \{1, \cdots, N\}$, *among which tasks* τ_k, $(k \in \mathcal{FT})$

are the FT-enabled tasks, $\mathcal{FT} \subseteq \{1, \cdots, N\}$. *If the task set* $\{\tau_i\}$ *is schedulable under the ORTGA task recovery and switch-back scheme with random failures, it is called (ORTGA)* **FT-schedulable**.

In this section, we will discuss the schedulability analysis when there is only one FT-enabled task in the task set, i.e. $\mathcal{FT} = \{k\}, 1 \leq k \leq N$. In ORTGA, when a fault in HPC is identified, the decision module will issue a recovery request (RR). At the same time, task τ_k^p will be aborted, and τ_k^a will be released for recovery. This potentially raises a mode-change problem, since tasks τ_k^p and τ_k^a usually have different timing parameters. We need to carry out schedulability analysis to guarantee that the tasks are schedulable during the recovery. Similarly, after the recovery, τ_k^a can be switched back to τ_k^p for performance considerations. We also need to guarantee the tasks are still schedulable with the switch-back. If a task set is schedulable under the ORTGA task recovery, it is called *FT-schedulable for recovery*. If a task set is schedulable under the ORTGA task switch-back, it is called *FT-schedulable for switch-back*.

Since the schedulability analysis methods are similar for both recovery and switch-back, we will only present the analysis for recovery, i.e. under the mode-change from τ_k^p to τ_k^a in the following discussion.

Schedulability Analysis for ORTGA Recovery Scheme

When the recovery request (RR) is initiated, the recovery procedure drives the system from the old operating mode (abbreviated as "old-mode") to the new operating mode (abbreviated as "new-mod"). The plant is controlled by the HPC in the old-mode and is controlled by the HAC in the new-mode correspondingly. Tasks in the old-mode are called old-mode tasks; while tasks in the new-mode are called new-mode tasks. In the following discussions, if we refer to task τ_k, we mean subtask τ_k^p when the context is in old-mode and subtask τ_k^a when the context is in new-mode.

The switching from τ_k^p to τ_k^a imposes transitional scheduling overhead, which may make

the whole task set unschedulable. In real-time analysis, this is called the mode-change problem [81,86]. In order to determine whether the task set is FT-schedulable under potential recoveries, we develop a schedulability test. It is based on a variant of the proposal presented by Real et al. in [86] with simplifications. The basic idea here is to consider each real-time task τ_i which may be affected by the transitional scheduling overhead in either old-mode or new-mode. Then we perform an offline response time analysis to test if it is schedulable in both modes.

When there is only one FT-enabled task τ_k in the task set, it is the only task which may initiate the mode-change. For other task τ_i, $(i \neq k)$, in order to maintain the highest periodicity, the task release pattern should not change before and after the mode-change. We call these tasks unchanged tasks.

The schedulability test is divided into 3 parts:

(I) Schedulability of Steady State Task Set

Let us define two task sets:

$$\mathcal{S}o \triangleq \{(C_1, T_1), \cdots, (C_k^p, T_k^p), \cdots, (C_N, T_N)\}; \tag{5.13}$$

$$\mathcal{S}n \triangleq \{(C_1, T_1), \cdots, (C_k^a, T_k^a), \cdots, (C_N, T_N)\}. \tag{5.14}$$

These are the old-mode and new-mode steady state task sets. We first test if both task sets are schedulable. This can be done using the standard response time analysis [58] for each task in the set. By solving the recurrence equations, we get R_i^{So} (and R_i^{Sn}), the maximal response time of task τ_i under the old-mode (and under the new-mode) in steady state. It should be smaller than or equal to the task deadline (period) T_i for schedulability. If the recurrence value exceeds the period, then task τ_i is unschedulable.

(II) Schedulability of Old-mode Tasks with Transitional Scheduling Overhead

In the old-mode, first, it is obvious to see that for each task τ_i, $(i < k)$, its schedulability is not affected by the mode-change. This is because they have higher priorities than the

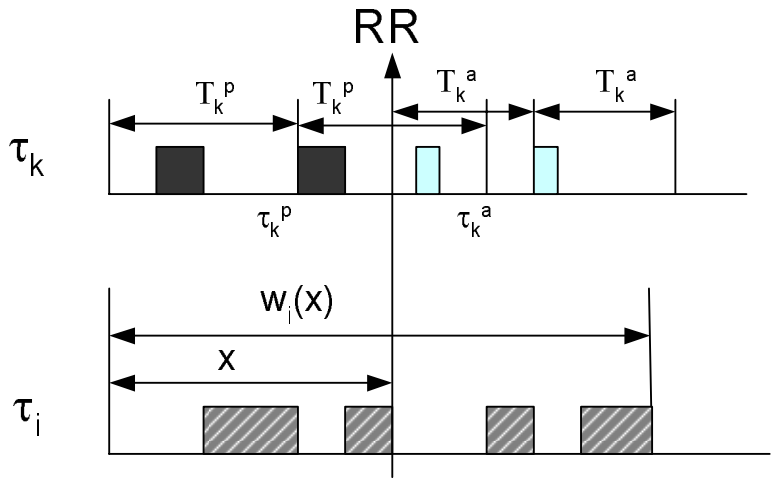


Figure 5.8: Illustration of mode-change incurred by the recovery.

FT-enabled task τ_k.

Secondly, we consider task τ_k^p, which is the task aborted upon the RR. It cannot be affected by the new-mode task τ_k^a during the old-mode, hence its schedulability has already been covered by the steady state schedulability analysis above (case (I)).

Lastly, we consider every task τ_i, $(i > k)$, which is an old-mode task who has lower priority than task τ_k. In order to account for the worst case phasing in the schedulability test, the RR is assumed to occur x time units after the task's activation [5]. We also define a temporal window $w_i(x)$, starting at the activation of old-mode task τ_i and finishing when τ_i completes. Figure 5.8 illustrates x and $w_i(x)$.

In the old-mode, τ_i first can be affected by the old-mode aborted task τ_k^p. The total worst case interferences from τ_k^p is:

$$\left\lfloor \frac{x}{T_k^p} \right\rfloor C_k^p + \min\left(x - \left\lfloor \frac{x}{T_k^p} \right\rfloor T_k^p, C_k^p\right). \tag{5.15}$$

τ_i can also be affected by the new released task τ_k^a. The interferences from τ_k^a is:

$$\left\lceil \frac{w_i(x) - x}{T_k^a} \right\rceil_0 C_k^a \quad , \tag{5.16}$$

[5] In other words, x is the phasing between the RR and activation time of τ_i

where $\lceil x \rceil_0 \triangleq \max\{\lceil x \rceil, 0\}$. Finally, τ_i can be affected by the unchanged tasks who have higher priorities than it. These interferences are:

$$\sum_{j<i, j\neq k} \left\lceil \frac{w_i(x)}{T_j} \right\rceil C_j \ . \tag{5.17}$$

By summing up all the sources of interferences, we got the total size of window

$$\begin{aligned} w_i(x) = C_i + \left\lfloor \frac{x}{T_k^p} \right\rfloor C_k^p + \min\left(x - \left\lfloor \frac{x}{T_k^p} \right\rfloor T_k^p, C_k^p\right) + \left\lceil \frac{w_i(x) - x}{T_k^a} \right\rceil_0 C_k^a \\ + \sum_{j<i, j\neq k} \left\lceil \frac{w_i(x)}{T_j} \right\rceil C_j \ . \end{aligned} \tag{5.18}$$

Solution to this equation is obtained by performing a recurrence calculation on $w_i(x)$ to find the smallest positive integer that satisfies it, just as the ordinary response time analysis. The response time of old-mode task τ_i is thus obtained as the duration of the largest time window of $w_i(x)$, i.e.,

$$R_i = \max(w_i(x)), \quad \forall x \in [0, R_i^{\mathcal{S}o}). \tag{5.19}$$

The significant values of x are those that produce changes in the values of the *ceiling* and *floor* functions in Equation (5.18).

(III) Schedulability of New-mode Tasks with Transitional Scheduling Overhead

In the new-mode, first, it is obvious to see that for each task i, $(i < k)$, its schedulability is not affected by the mode-change. So this case has already been covered in the steady state schedulability test (case (I)).

Secondly, we consider task τ_k^a, which is the newly released task at RR. It can not be affected by the old-mode task τ_k^p in the new-mode, hence its schedulability has also been covered by the steady state schedulability analysis (case (I)).

Finally, we consider every task τ_i, $(i > k)$, which is a new-mode unchanged task who has

lower priority than task τ_k. As with the analysis of old-mode tasks, we can define a temporal window w_i to enclose the response time.

In the new-mode, τ_i can be affected by the new task τ_k^a. The interference is:

$$\left\lceil \frac{w_i}{T_k^a} \right\rceil C_k^a \quad . \tag{5.20}$$

Also, it can be affected by the unchanged tasks whose priorities are higher. These interferences are:

$$\sum_{j<i, j\neq k} \left(\left\lceil \frac{w_i}{T_j} \right\rceil C_j \right) \quad . \tag{5.21}$$

Summing up all the sources of interferences, we get the window size

$$w_i = C_i + \left\lceil \frac{w_i}{T_k^a} \right\rceil C_k^a + \sum_{j<i, j\neq k} \left(\left\lceil \frac{w_i}{T_j} \right\rceil C_j \right) \quad . \tag{5.22}$$

Again, recurrence calculation of w_i should be performed until its convergence or when the recurrence value exceeds the task's deadline. Then the response time for task τ_i in the new-mode can be obtained as:

$$R_i = \max(w_i, R_i^{\mathcal{S}n}). \tag{5.23}$$

The whole task set $\{\tau_1, \cdots, \tau_N\}$ is schedulable even with random recoveries if it passes the schedulability tests (I-III), i.e. FT-schedulable for recovery. Similar tests can be done to ensure the task set is still schedulable with the switch-back.

5.5.4 Co-design of Fault Tolerance and Scheduling

As we have shown in Section 5.5.2, the shorter the HAC's period, the larger the maximum stability region. A larger maximum stability region (MSR) is desirable from the fault tolerance point of view, because it minimizes the unduly restriction of the state space the HPC can use. However, from the perspective of scheduling, we can not use too small an HAC

period, since this will make the whole task set unschedulable. So a natural question is *"What is the optimal HAC period such that the HAC's MSR is maximized and the system is still schedulable?"*. We call this the fault tolerance and scheduling co-design problem.

For multiple FT-enabled tasks under the protection of ORTGA, the fault tolerance and scheduling co-design problem can be formally expressed as:

Problem 3. $Maximize_{\{T_k^a\}_{k\in\mathcal{FT}}} \quad \sum_{k\in\mathcal{FT}} A(T_k^a)$

s.t.: Task set is FT-schedulable.

Here $\mathcal{FT}$ is the set of all FT-enabled tasks and $A(T_k^a)$ is the area of the MSR corresponding to task τ_k with control loop period T_k^a.

When there is only one FT-enabled task, the co-design problem can be reformulated as follows:

Problem 4. *Given the task set $\{\tau_i\}$, $i \in \{1, \cdots, N\}$, among which task τ_k is the FT-enabled task. (C_i, T_i) are known parameters for the real-time tasks τ_i, $(i \neq k)$. For task τ_k, the timing parameters (C_k^p, T_k^p) and C_k^a are known for the HPC and HAC respectively. Find the minimum possible period T_k^{*a} of the HAC such that the task set is FT-schedulable.*

This is because when there is only one FT-enabled task, since $A(T)$ is a decreasing function of T, the minimum possible period T_k^{*a} which makes task set FT-schedulable is the solution to Problem 3.

Remember FT-schedulable requires the task set is schedulable under both recovery and switch-back. To find the solution to Problem 4 under the recovery case, we can use the binary search algorithm [54]. In the binary search, during each recursion, we test if the current task set is FT-schedulable for recovery using the schedulability test algorithm we have presented in Section 5.5.3, until we found $T_k'^a$ which is the minimum period to make the task set schedulable for recovery. After $T_k'^a$ is obtained, in general, we should continue to test whether this period also makes the task set schedulable for switch-back. If $T_k'^a$ indeed passes the switch-back test, we know $T_k^{*a} = T_k'^a$. However, in practice, even if $T_k'^a$ does

not pass the test, we do not bother to search for larger period that can make the task set both schedulable for recovery and switch-back. There are two reasons for this: 1) a larger period than $T_k^{'a}$ means smaller maximum stability region of the HAC, which is undesirable; 2) unlike the recovery, the switch-back is not an emergent event. So a practical approach to initiate the switch-back is to wait until when both CPU meets an idle interval and the plant state is near the control set point. The former criteria enforces there is no effect of the mode-change on the schedulability associated with the switch-back. The latter criteria guarantees the plant is stable with the switch-back.

The use of binary search algorithm for optimal period selection requires an ordering property to be held. To this end, we give the following Lemma.

Lemma 5.5.2. *If $T_k^a = x > 0$ makes the task set FT-schedulable for recovery under our schedulability test presented in Section 5.5.3, then for any $T_k^a = y$, where $y > x$, the task set will also be FT-schedulable for recovery under our schedulability test.*

Proof. With a larger value for the T_k^a and the same value for C_k^a, for any time window, the new task τ_k^a only generates less interference in the old-mode and new-mode for the recovery schedulability test (see Equation (5.18) and (5.22)). So if $T_k^a = x > 0$ makes the task set FT-schedulable for recovery under our schedulability test, for $T_k^a = y$, where $y > x$, the task set will also be FT-schedulable for recovery under our schedulability test. □

Note most of the time, we have $C_k^a < C_k^p$, thus we can have $T_k^{*a} < T_k^p$, where T_k^{*a} is the minimum period of HAC such that the task set is FT-schedulable for recovery. Below we show a numerical example based on the results of the binary search algorithm.

Example 5.5.1. *Consider three tasks τ_1, τ_2, τ_3. Task τ_1 and τ_3 are ordinary real-time tasks with timing parameters (2, 4) and (3, 30) respectively. Task τ_2 is an FT-enabled task with timing parameters (2, 8) for the HPC. Then we have the following result:*

- *If $C_2^a = 2.0$, we have $T_2^{*a} = 6.5 < T_2^p$;*

- *If* $C_2^a = 1.5$*, we have* $T_2^{*a} = 4.5 < T_2^p$;
- *If* $C_2^a = 1.0$*, we have* $T_2^{*a} = 3.0 < T_2^p$;
- *If* $C_2^a = 0.5$*, we have* $T_2^{*a} = 2.5 < T_2^p$.

We also see that as the decreasing of C_2^a, the optimal T_2^{*a} has significantly dropped. This means the HAC can run at a faster control rate during the recovery when fault occurs.

5.6 Fault Tolerance and Scheduling Co-Design – Multiple FT-enabled Tasks Case

When there are more than one FT-enabled tasks in the task set, the co-design of fault tolerance and scheduling problem becomes extremely difficult to solve. This is because with multiple plants under the control of possible faulty HPCs, each controller may fail independently. If the HPC for plant k (denoted as HPC_k) is currently under the recovery, at the same time the HPC for another plant m (denoted as HPC_m) also fails, this will cause *concurrent mode-changes*. We need to provide an algorithm for schedulability test when concurrent mode-changes could occur. However, this is an open research issue. Most research on schedulability analysis of mode-changes precludes the occurrence of concurrent mode-changes [86]. Yet, in practice, we need to provide fault tolerance support for multiple plants who may fail at random time. It is desirable to have a solution such that the system is FT-schedulable, at the same time, maximizes the total fault coverage. Though it seems formidable at first glance, in this section, we provide a practical solution to the co-design problem with possible concurrent random failures.

5.6.1 A Baseline Scheme for Multiple FT-enabled Tasks Case

The idea of our approach is trying to remove the effect of concurrent mode-changes. So the schedulability analysis can be greatly simplified or reduced to what we have already

discussed in the single FT-enabled task case. To this end, we first present a baseline recovery and switch-back scheme such that there are no concurrent mode-changes in this subsection. Based on this baseline scheme, in next subsection (Section 5.6.2) we extend it to a hybrid scheme for better fault coverage.

Note for every FT-enabled task τ_k, $(k \in \mathcal{FT})$, usually we have $C_k^p > C_k^a$, since the HPC performs more complex computations in order to achieve higher performance while the HAC's algorithm tends to be simpler and less computationally intensive. Let's expand the worst case execution time of both τ_k^p and τ_k^a to $C_k^{'} \triangleq \max\{C_k^p, C_k^a\}$. We also select $T_k^a = T_k^p$ for the HAC in the baseline scheme. Whenever a fault occurs in τ_k^p, the decision module aborts the faulty task τ_k^p, but wait until the end of current period to release the new task τ_k^a. Then for all other tasks τ_j, $(j \neq k)$, task τ_k just looks like an ordinary periodic task, with worst case execution time $C_k^{'}$ and period $T_k^{'} \triangleq T_k^p = T_k^a$. No matter when a fault occurs, there is no mode-change effect to other tasks. We refer this new scheme as the *same-period scheme*, as compared with the previous discussed ORTGA recovery scheme. The previous scheme uses $T_k^a \neq T_k^p$, hence we refer it as *diff-period scheme*. If every FT-enabled task τ_k is implemented using the *same-period scheme*, the fault recovery and switch-back can be done independently at random time, without mode-change incurred scheduling overheads to other tasks. As a result, there is no need to perform schedulability analysis of mode-changes.

However, there are two drawbacks of the *same-period scheme* as compared to the *diff-period scheme*:

1. Selecting $T_k^a = T_k^p$ may unduly shrink the HAC-established maximum stability region as shown in Example 5.5.1;

2. In the *same-period scheme*, the worst-case recovery delay, i.e. the time from the last valid control command is actuated to the new control takes effect is $T_k^p + T_k^a = 2 \cdot T_k^p$. While in the *diff-period scheme*, the worst-case recovery delay is $T_k^p + T_k^{*a}$, smaller than that of the same-period scheme. This implies using diff-period scheme will give

a smaller disturbance during the recovery.

Despite the above-mentioned drawbacks, the introduction of same-period scheme achieves the benefits that each task can independently recover and switch-back without ever affecting the schedulability of other tasks. We use this as a baseline recovery and switch-back scheme when multiple FT-enabled tasks exist.

5.6.2 A Hybrid Scheme for Multiple FT-enabled Tasks Case

The baseline recovery and switch-back scheme does not try to maximize the total maximum stability region. It only takes the scheduling perspective into account, but does not consider the fault tolerance requirements. In this subsection, we present a hybrid recovery and switch-back scheme which gives better performance leveraging the co-design of maximum stability region and scheduling.

In this new scheme, for each FT-enabled task τ_k, $(k \in \mathcal{FT})$, we first independently find the minimum period T_k^{*a} for the HAC can use, treating all other FT-enabled tasks as ordinary real-time tasks, using the method presented in Section 5.5.4 for single FT-enabled task case. T_k^{*a} imposes an lower bound of the HAC control loop period that can be used for recovery.

Now for each for each FT-enabled task τ_k, $(k \in \mathcal{FT})$, we have two MSR regions corresponding to the HAC's two control period choices — T_k^p and T_k^{*a}. Let us denote the MSR of HAC when control period is T_k^p as R_k^p. Similarly, we denote the MSR of HAC when control period is T_k^{*a} as R_k^{*a}. This is illustrated in Figure 5.9. Generally $T_k^{*a} < T_k^p$, so we know the area of R_k^{*a} is larger than the area of R_k^p. Otherwise, we set the HAC's control period the same as T_k^p, i.e. use the baseline recovery scheme for this task to avoid the possible mode-change in scheduling.

In the hybrid scheme, during the runtime, at first, all FT-enabled plants are under the control of their HPCs. Whenever the first time a plant's state (say plant k) is outside its cor-

responding R_k^p (the HAC-established maximum stability region corresponding to period T_k^p), this plant begin using R_k^{*a} (the HAC-established maximum stability region corresponding to period T_k^{*a}) for fault recovery switching decision. At the same time, all other FT-enabled plants j ($j \in \mathcal{FT}, j \neq k$) begin using R_j^ps for their fault recovery switching decision until plant k's state is back to its inner maximum stability region R_k^p.

This means the first plant whose state is out of its inner maximum stability region will use the diff-period scheme for recovery. Other plants will use the same-period scheme for recovery. This hybrid scheme will ensure that at any time, there is at most one recovery subtask τ_k^a running at the period of T_k^{*a}. Hence there is at most one mode-change occurring at any time. From the offline design of T_k^{*a}, we know the task set is always schedulable because this reduces multiple FT-enabled tasks case to single FT-enabled task case. As a result, we know the hybrid scheme makes all plants stable during the recoveries when faults occur, at the same time still maintains the whole task set to be schedulable.

In practice, due to the digital implementation of controllers, the decision of whether a plant's state is outside the HAC-established MSR is also enforced at discrete time. In the rare scenario, two or more plants' state may be outside the MSR of their R_k^ps at the same time-intersected decision epoch. When this happens, these plants all want to use the diff-period scheme, which is not viable. To preclude such situation from occurring, at any decision time t for any plant k, we use two-periods state projections to decide whether to initiate the diff-period scheme or not. This means that if the projected state $x_k(t + 2T_k^p)$ is outside its R_k^p, this plant will begin using the diff-period scheme, making other FT-enabled plants use the same-period scheme. If there are more than two plants' (say plant k and plant j) projected states are outside their corresponding inner MSRs (R_k^p and R_j^p), then we broke the tie by selecting one of them to begin using the diff-period scheme, and other FT-enabled plants all begin using the same-period scheme, as illustrated in Figure 5.9. Since we plan two periods ahead, all the plants can still maintain stability after the first selected plant begin using the diff-period scheme. This is because for any FT-enabled plant k, only two

periods later at time $t + 2T_k^p$, it could be outside of its R_k^p, the HAC-established MSR using T_k^p. So at one period later, i.e. at time $t + T_k^p$, it is still within R_k^p, hence all these plants are stable using the the same-period scheme.

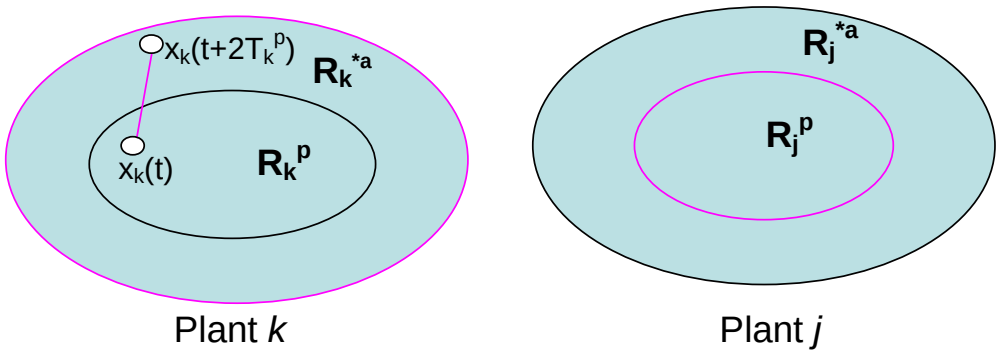


Figure 5.9: Illustration of the hybrid recovery scheme for multiple FT-enabled tasks case.

5.7 Conclusions

In this chapter we propose a new fault tolerance architecture, ORTGA, for real-time control systems. ORTGA delivers the same functionalities as Simplex, with the same high fault coverage and reliability. Compared with Simplex, ORTGA has advantages including that it allows more efficient resource utilizations and more flexibility. This is achieved by running the high-assurance controller in an on-demand fashion instead of running in parallel. ORTGA is applicable for most industrial applications where both efficient resource usage and high fault coverage are desired.

Based on the design and analysis of ORTGA, we also proposed the research on fault tolerance and scheduling co-design problem in real-time control systems and presented its practical solutions.

Chapter 6

Final Remarks

Performance management and fault tolerance are two important issues faced by computing systems research. This dissertation investigates these two issues through theoretical modeling, analysis, architecture design, and system development. We establish novel approaches exploiting feedback control for both performance management and fault tolerance.

In the first part of this dissertation, we propose Queueing Model Based Feedback Control framework to regulate system performance under highly dynamic environments via adaptive resource allocation. Traditionally, queueing theory was used for modeling computing system's performance. It usually serves as an offline capacity planning tool. On the other hand, feedback control theory was successfully used for dynamically controlling the performance of electro-mechanical systems. How to utilize the "descriptive" power of queueing theory and the "prescriptive" power of feedback control to control computing system's performance was an open problem. Queueing Model Based Feedback Control answers this question by integrating the strengths of both queueing models and feedback control into one unified framework. In this framework, queueing model's online solution provides a feed forward prediction. It keeps the system state near an equilibrium operation point, in spite of abrupt changes in the traffic. The use of feedback control augments the queueing model feed forward control. It helps to correct the residual errors due to model inaccuracies and measurement errors for real-life systems.

We have conducted extensive experiments using Queueing Model Based Control in managing the timing behavior of different real-life applications. In this dissertation, we use Apache Web server and multi-tiered Web application to demonstrate the efficacy of Queue-

ing Model Based Feedback Control framework. Our experiments showed that this elegant framework achieves better performance regulation than previous proposals. It is intellectually satisfying to see that two separately developed powerful theories can work synergistically in a unified framework.

Queueing Model Based Feedback Control also has the advantage of general applicability to a wide range of computing systems. By exploiting different combinations of queueing models and feedback control techniques, we can get a family of schemes within our framework suited to different computing applications and different QoS goals.

In the second part of this dissertation, we further exploit the use of feedback control to achieve fault tolerance for real-time embedded control systems. Specifically, we propose ORTGA (On-demand Real-Time GuArd), a new fault tolerance architecture. ORTGA uses state feedback to control software execution. Together with rigorous scheduling analysis, it ensures system safety by real-time fault recovery. ORTGA delivers the same functionalities and high fault coverage as previously proposed Simplex architecture. Further it has the advantages of efficient resource utilizations and allows much more flexibility for control and fault tolerance design. ORTGA can be deployed in a wide range of real-time embedded applications where both an efficient resource utilization and a high fault coverage are desired.

We implemented ORTGA prototype in an inverted pendulum testbed to demonstrate its efficacy and efficiency. Based on the ORTGA design, we discussed the fault tolerance and scheduling co-design problem and presented its solutions.

Feedback is a universal mechanism which exists in many disciplines. During the last 50 years, feedback control theory has been very successful in industry for controlling electro-mechanical systems. While this dissertation explores two new aspects on using feedback control to solve issues of computing systems, focusing on performance management and fault tolerance, we believe feedback control is a powerful tool which can help us in solving many other issues related to computing systems as well. These include system scalability, reliability, security, and evolvobility. All these areas are well-deserved to be further exploited.

Bibliography

[1] National Research Council. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers.* National Academies Press, 2001.

[2] Abdelzaher, T., He, T., and Stankovic, J. Feedback control of data aggregation in sensor networks. In *43rd IEEE Conference on Decision and Control* (Paradise Island, Bahamas, 2004).

[3] Abdelzaher, T., Shin, K. G., and Bhatti, N. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems 13*, 1 (2002).

[4] Abdelzaher, T. F., and Lu, C. Modeling and performance control of internet servers. In *39th IEEE Conference on Decision and Control* (Sydney, Australia, 2000).

[5] ACM. ACM sigmetrics performance evaluation review. *ACM SIGMetrics Performance Evaluation Review* (1972-2005).

[6] Adler, S. The slashdot effect, an analysis of three internet publications. *Linux Gazette* (1999).

[7] Arlitt, M., and Jin, T. 1998 world cup web site access logs. `http://www.acm.org/sigcomm/ITA/`, 1998.

[8] Astrom, K. J., and Hagglund, T. *PID Controllers: Theory, Design, and Tuning.* Instrument Society of America, 1995.

[9] Astrom, K. J., and Wittenmark, B. *Adaptive Control (2nd Edition).* Prentice Hall, 1994.

[10] Astrom, K. J., and Wittenmark, B. *Computer-Controlled Systems: Theory and Design, 3rd edition.* Addison-Wesley Pub Co., 1994.

[11] Avizienis, A. The methodology of n-version programming. In *Software Fault Tolerance*, M. R. Lyu, Ed. John Wiley and Sons, New York, 1995.

[12] Aydin, H., Melhem, R., and Mosse, D. Optimal scheduling of imprecise computation tasks in the presence of multiple faults. In *Seventh International Conference on Real-Time Computing Systems and Applications (RTCSA00)* (2000).

[13] Aydin, H., Melhem, R., and Mosse, D. Tolerating faults while maximizing reward. In *Twelfth Euromicro Conference on Real-Time Systems (Euromicro'00)* (Stockholm, Sweden, 2000).

[14] Barry, D. K. *Web Services and Service-Oriented Architectures: The Savvy Manager's Guide*. Morgan Kaufmann, 2003.

[15] Bezenek, T., Cain, T., Dickson, R., Heil, T., Martin, M., McCurdy, C., Rajwar, R., Weglarz, E., Zilles, C., and Lipasti, M. Characterizing a java implementation of TPC-W. In *The 3rd Workshop On Computer Architecture Evaluation Using Commercial Workloads (CAECW)* (Toulouse, France, 2000), http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[16] Blumenfeld, D. *Operations Research Calculations Handbook*. CRC Press, 2001.

[17] Boyd, S., Ghaoui, L. E., Feron, E., and Balakrishnan, V. *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics (SIAM), 1994.

[18] Cady, J., and Howarth, B., Eds. *Computer systems performance management and capacity planning*. Prentice-Hall, Inc., 1990.

[19] Candea, G. Enemies of dependability I: Software. `www.stanford.edu/~candea/teaching/cs444a-fall-2003/notes/software.pdf`, 2003.

[20] census.gov. Quarterly retail e-commerce sales, 3rd quarter 2004, 2004.

[21] Chadd, A., Collins, R., Nordstrom, H., Rousskov, A., and Wessels, D. Squid web proxy cache, 2005.

[22] Chandra, R., Liu, X., and Sha, L. On the scheduling of flexible and reliable real-time control systems. *Real-Time Syst. 24*, 2 (2003), 153–169.

[23] Chen, H., and Mohapatra, P. Session-based overload control in qos-aware web servers. In *IEEE INFOCOM* (2002).

[24] Chen, X., Mohapatra, P., and Chen, H. An admission control scheme for predictable server response time for web accesses. In *10th international conference on World Wide Web* (Hong Kong, 2001).

[25] Cherkasova, L., and Phaal, P. Session-based admission control: A mechanism for peak load management of commercial web sites. *IEEE Trans. Comput. 51*, 6 (2002), 669–685.

[26] Cristian, F. Understanding fault-tolerant distributed systems. *Communications of the ACM 34*, 2 (1991), 56–78.

[27] Crovella, M., and Bestavros, A. Self-similarity in world wide web traffic: Evidence and possible cause. In *ACM SIGMETRICS* (Philadelphia, Pennsylvania, 1996).

[28] Crovella, P. B., and Mark. Generating representative web workloads for network and server performance evaluation. In *Measurement and Modeling of Computer Systems* (1998), pp. 151–160.

[29] Diao, Y., Gandhi, N., Hellerstein, J. L., Parekh, S., and Tilbury, D. M. Mimo control of an apache web server: Modeling and controller design. In *American Control Conference* (2002).

[30] Diao, Y., Hellerstein, J., and Parekh, S. Using fuzzy control to maximize profits in service level management. *IBM Syst. J., vol. 41, no. 3, pp. 403-420, 2002.* (2002).

[31] Franklin, G. F., Powell, D. J., and Workman, M. L. *Digital Control of Dynamic Systems (3rd Edition).* Prentice Hall, 1997.

[32] Gandhi, N., Parekh, S., Hellerstein, J., and Tilbury, D. Feedback control of a lotus notes server: Modeling and control design. In *American Control Conference* (2001).

[33] Goodwin, G. C., and Sin, K. S. *Adaptive Filtering Prediction and Control.* Prentice-Hall, 1984.

[34] Graham, S., Baliga, G., and Kumar, P. R. Issues in the convergence of control with communication and computing: proliferation, architecture, design, services, and middleware. In *Proceedings of the 43rd IEEE Conference on Decision and Control* (December 2004).

[35] Hellerstein, J. L., Diao, Y., Parekh, S., and Tilbury, D. M. *Feedback Control of Computing Systems.* Wiley-IEEE Press, 2004.

[36] Herbert, G. W. Failure from the field: Complexity kills. In *Second Workshop on Evaluating and Architecting System dependabilitY (EASY)* (San Jose, CA, 2002).

[37] Hill, B. J., and Thomson, K. System performance management: Moving from chaos to value. Tech. rep., Sun Microsystems, 2001.

[38] ISI. The network simulator. `http://www.isi.edu/nsnam/ns`.

[39] Jacobson, V. Congestion avoidance and control. In *ACM SIGCOMM'88* (1988), pp. 314–329.

[40] Jain, R. *The Art of Computer Systems Performance Analysis.* John Wiley & Sons, 1991.

[41] Jin, D. M., and Tai. httperf: A tool for measuring web server performance. In *First Workshop on Internet Server Performance* (1998).

[42] Jin, L., Machiraju, V., and Sahai, A. Analysis on service level agreement of web services. Tech. Rep. HPL-2002-180, HP Labs, 2002.

[43] Kaes, R. J., and Young, S. tinyproxy, 2005.

[44] Kamra, A., Misra, V., and Nahum, E. Yaksha: A controller for managing the performance of 3-tiered websites,. In *The Twelfth IEEE International Workshop on Quality of Service (IWQoS 2004)* (Toronto, Canada, 2004).

[45] Kanodia, V., and Knightly, E. W. Ensuring latency targets in multiclass web servers. *IEEE Transactions on Parallel and Distributed Systems* (2003).

[46] Karlsson, M., Karamanolis, C., and Zhu, X. Triage: Performance isolation and differentiation for storage systems. In *The Twelfth IEEE International Workshop on Quality of Service (IWQoS 2004)* (2004).

[47] Kernighan, B. W., and Ritchie, D. *The C Programming Language (2nd Edition)*. Prentice Hall PTR, 1988.

[48] Kim, K. H. K. Slow advances in fault-tolerant real-time distributed computing. In *Proceedings of 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)* (Florianpolis, Brazil, October 2004), pp. 106–108.

[49] Klein, M., Ralya, T., Pollak, B., Obenza, R., and Harbour, M. G. H. *A Practitioner's Handbook for Real-Time Analysis: Guide to RateMonotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, 1993.

[50] Kleinrock, L. *Queueing Systems: Vol 1: Theory*. John Wiley, 1975.

[51] Kleinrock, L. *Queueing Systems, Vol. 2, Applications*. John Wiley, 1976.

[52] Lazowska, E. D., Zahorjan, J., Graham, G. S., and Sevcik, K. C., Eds. *Quantitative system performance: computer system analysis using queueing network models*. Prentice-Hall, Inc., 1984.

[53] Lee, K., and Sha, L. Process resurrection: A fast recovery mechanism for real-time embedded systems. In *11th IEEE Real-Time and Embedded Technology and Applications Symposium* (San Francisco, California, 2005).

[54] Leiserson, T. H., Cormen, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[55] Levine, W. S. *The Control Handbook*. CRC Press, 1996.

[56] Li, B., and Nahrstedt, K. A control-based middleware framework for quality-of-service adaptations. *IEEE Journal on Selected Areas in Communications 17*, 9 (1999), 1632–1650.

[57] Liu, C. L., and Layland, J. W. Scheduling algorithms for multiprogramming in hard real time environment. *Journal of the ACM. 20*, 1 (1973), 40C61.

[58] Liu, J. *Real-Time Systems*. Prentice Hall PTR, 2000.

[59] Liu, X., Ding, H., Lee, K., Sha, L., and Caccamo, M. Feedback based real-time fault tolerance – issues and possible solutions. In *First International Workshop on Feedback Control Implementation and Design in Computing Systems and Networks (FeBID 2006), co-located with NOMS 2006* (Vancouver, Canada, 2006).

[60] Liu, X., Heo, J., and Sha, L. Modeling 3-tiered web applications. In *13th Annual Meeting of the IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS 2005), short paper* (Atlanta, Georgia, 2005).

[61] Liu, X., Heo, J., Sha, L., and Zhu, X. Adaptive control of multi-tiered web application using queueing predictor. In *10th IEEE/IFIP Network Operations and Management Symposium (NOMS 2006)* (Vancouver, Canada, 2006).

[62] Liu, X., Zheng, R., Heo, J., Wang, Q., and Sha, L. Timing control for web server systems using internal state information. In *International Conference on Networking and Services (ICNS 2005)* (Tahiti, French Polynesia, 2005).

[63] Liu, X., Zhu, X., Singhal, S., and Arlitt, M. Adaptive entitlement control to resource containers on shared servers. In *9th IFIP/IEEE International Symposium on Integrated Network Management (IM 2005)* (Nice, France, 2005).

[64] Ljung, L. *System Identification: Theory for the User (2nd Edition)*. Prentice Hall PTR, 1998.

[65] Lofberg, J. YALMIP : A toolbox for modeling and optimization in matlab. In *2004 IEEE International Symposium on Computer Aided Control Systems Design* (Taipei, Taiwan, 2004).

[66] Lu, C., Abdelzaher, T., Stankovic, J., and Son, S. A feedback control approach for guaranteeing relative delays in web servers. In *IEEE Real-Time Technology and Applications Symposium* (TaiPei, Taiwan,, 2001).

[67] Lu, Y., Abdelzaher, T., Lu, C., Sha, L., and Liu, X. Feedback control with queueing-theoretic prediction for relative delay guarantees in web servers. In *9th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE Computer Society, 2003.

[68] Lu, Y., Abdelzaher, T. F., and Tao, G. Direct adaptive control of a web cache system. In *American Control Conferencerence* (Denver, CO, 2003).

[69] Lu, Y., Lu, C., Abdelzaher, T., and Tao, G. An adaptive control framework for qos guarantees and its application to differentiated caching services. In *IWQoS* (Miami Beach, FL, 2002).

[70] Lu, Y., Saxena, A., and Abdelzaher, T. F. Differentiated caching services: A control-theoretical approach. In *International Conference on Distributed Computing Systems* (Phoenix, Arizona, 2001).

[71] Lyu, M., Ed. *Software Fault Tolerance (Trends in Software, No. 3).* John Wiley and Sons, New York, 1995.

[72] MacDougall, M. H. *Simulating Computer Systems: Techniques and Tools.* MIT Press Series in Computer Systems. MIT Press, 1987.

[73] madbean.com. Apache foundations exponential rise. `http://madbean.com/2005/mb2005-11/`, 2005.

[74] Melliar-Smith, P. M., and Moser, L. E. Progress in real-time fault tolerance. In *Proceedings of 23rd IEEE International Symposium on Reliable Distributed Systems (SRDS'04)* (Florianpolis, Brazil, October 2004), pp. 109–111.

[75] Menasce, D., and Almeida, V. *Capacity Planning for Web Services: Metrics, Models, and Methods.* Prentice Hall PTR, 2001.

[76] Mogul, J. C. Operating systems support for busy internet servers. In *Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)* (Orcas Island, WA, 1995).

[77] MySQL AB. Mysql. `http://www.mysql.com/`.

[78] Netcraft. February 2006 web server survey. `http://news.netcraft.com/archives/2006/02/02/february_2006_web_server_survey.html`, 2006.

[79] Ogata, K. *Modern Control Engineering, 3rd edition.* Prentice Hall, 1996.

[80] Parekh, S., Gandhi, N., Hellerstein, J. L., Tilbury, D., Jayram, T. S., and Bigus, J. Using control theory to achieve service level objectives in performance management. *Real Time Systems Journal 23*, 1-2 (2001).

[81] Pedro, P., and Burns, A. Schedulability analysis for mode changes in real-time systems. In *10th Euromicro workshop on Real-Time systems* (Berlin, Germany, 1998).

[82] Pradhan, D., and Vaidya, N. Roll-forward checkpointing scheme: A novel fault-tolerant architecture. *IEEE Transactions on Computers 43*, 10 (October 1994).

[83] Rajagopalan, M., Debray, S. K., Hiltunen, M. A., and Schlichting, R. D. Profile-directed optimization of event-based programs. In *SIGPLAN '02 Conf. on Prog. Language Design and Implementation (PLDI 02),* (2002).

[84] Ramakrishnan, R., and Gehrke, J. *Database Management Systems (3rd Edition).* McGraw-Hill, 2003.

[85] Randell, B., and Xu, J. The evolution of the recovery block concept. In *Software Fault Tolerance.* John Wiley and Sons, 1995.

[86] Real, J., and Crespo, A. Mode change protocols for real-time systems: A survey and a new proposal. *Real-Time Systems 26* (2004), 161–197.

[87] Real-Time System Services Working Group. IEEE STD 1003.1, 1996 edition, 1998.

[88] Seto, D., Ferreira, E., and Marz, T. Case study: Development of a baseline controller for automatic landing of an F-16 aircraft using lmis. Tech. Rep. CMU/SEI-99-TR-020, Software Engineering Institute, Carnegie Mellon University, 2000.

[89] Seto, D., Krogh, B. H., Sha, L., and Chutinan, A. Dynamic control system upgrade using the simplex architecture. *IEEE Control System Magazine* (1998).

[90] Seto, D., Lehoczky, J. P., Sha, L., and Shin, K. G. On task schedulability in real-time control system. In *Proceedings of the 17th IEEE Real-Time Systems Symposium* (1996), pp. 13–21.

[91] Seto, D., and Sha, L. An engineering method for safety region development. Tech. Rep. 18, CMU SEI, 1999.

[92] Sha, L. Dependable system upgrade. In *RTSS '98: Proceedings of the IEEE Real-Time Systems Symposium*. IEEE Computer Society, 1998, p. 440.

[93] Sha, L., Liu, X., Caccamo, M., and Buttazzo, G. Online control optimization using load driven scheduling. In *Conference on Decision and Control* (Sydney, Australia, 2000).

[94] Sha, L., Liu, X., Lu, Y., and Abdelzaher, T. Queueing model based network server performance control. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. IEEE Computer Society, 2002, p. 81.

[95] Sha, L., Rajkumar, R., and Sathaye, S. Generalized rate monotonic scheduling theory: A framework of developing real-time systems. *IEEE Proceedings* (1994).

[96] Stankovic, J. A. Real-time computing systems: The next generation. In *Tutorial: Hard Real-Time Systems*, J. A. Stankovic and K. Ramamritham, Eds. IEEE Computer Society, 1998, pp. 14–37.

[97] Stankovic, J. A., Lu, C., Son, S. H., and Tao, G. The case for feedback control real-time scheduling. In *11th EuroMicro Conference on Real-Time Systems* (York, UK, 1999).

[98] The Apache Group. Apache http server project. `http://httpd.apache.org/`, 2004.

[99] The Apache Jarkata Project Group. Tomcat, 2005.

[100] The Transaction Processing Council. TPC-W, 2002.

[101] UDDI. Universal description, discovery, and integration of business for the web, 2004.

[102] Walpole, D. C. S., Goel, A., Gruenberg, J., McNamee, D., Pu, C., and Jonathan. A feedback-driven proportion allocator for real-rate scheduling. In *Operating Systems Design and Implementation* (1999), pp. 145–158.

[103] Welsh, M., and Culler, D. Adaptive overload control for busy internet servers. In *4th USENIX Symposium on Internet Technologies and Systems (USITS '03)* (Seattle, WA, 2003).

[104] WILSON, T. The cost of downtime. `http://www.internetweek.com/lead/lead073099.htm`, 1999.

[105] Wu, S.-P., and Boyd, S. Sdpsol: a parser/solver for sdp and maxdet problems with matrix structure. In *Recent Advances in LMI Methods for Control*, L. E. Ghaoui and S.-I. Niculescu, Eds. SIAM, 1999.

[106] Zhang, R., Lu, C., Abdelzaher, T. F., and Stankovic, J. A. Controlware: A middleware architecture for feedback control of software performance. In *International Conference on Distributed Computing Systems* (Vienna, Austria, 2002.).

Author's Biography

Xue Liu is a PhD in computer science from the University of Illinois at Urbana-Champaign. He has broad research interests in real-time and embedded systems, performance management of networked server systems, sensor networks, fault tolerance and reliability. He received his Bachelor of Science degree in Applied Mathematics and Master of Engineering degree in Control Theory and Applications, both from Tsinghua University, Beijing, China in 1996 and 1999 respectively. During his PhD study, he has also worked as research summer interns in IBM T. J. Watson Research Center in Hawthorne, NY and HP Labs in Palo Alto, CA. He received the Ray Ozzie Fellowship, the Saburo Muroga Fellowship, the C. W. Gear Outstanding Graduate Award, and the Mavis Memorial Fund Scholarship Award from University of Illinois at Urbana-Champaign.